

I.

Dr. Úry László

COMMODORE 64

LSI Alkalmazástechnikai
Tanácsadó Szolgálat

BÉRSZABÁLYOZÁS!

c. BASIC nyelvű COMMODORE 64 gépre kidolgozott program

A BÉRSZABÁLYOZÁS c. program a vonatkozó rendelkezések szerint az alábbi négy

- központi A
- központi B
- jövedelmezőségtől függő (eszközbér arányos)
- saját döntés alapján központi

bérszabályozási formára -- a rendelkezésekben előírt módon és input adatokból kiindulva – adott bérfejlesztés, és adott nettó R-alap mellett számítja a

1. PROGRESSZÍV NYERESÉGADÓT és a BRUTTÓ R-ALAP IGÉNYT

vagy

Adott bruttó R-alap és adott nettó R-alap mellett keresi a

2. MEGVALÓSÍTHATÓ BÉRFEJLESZTÉST és a PROGRESSZÍV NYERESÉGADÓT

illetve

Adott bruttó R-alap és adott bérfejlesztés mellett keresi a

3. NETTÓ R-ALAPOT és a PROGRESSZÍV NYERESÉGADÓT

illetve számítja a

4. LEHETSÉGES ADÓMENTES BÉRSZÍNVONALAT és az ADÓMENTES BÉRSZÍNVONALNÖVELESI LEHETŐSÉG %-át

A program forgalmazója: ECONOMIX Közgazdász Egyetemi Kiszövetkezet. 1828 Bp. 5. Pf. 489.

Ügyintéző: Forrai István Tel.: 174-647
Csák Csabáné Tel.: 174-031

A program jelenleg: BASIC nyelven
COMMODORE 64 és a
SHARP PC 1500 személyi számítógépekre
áll rendelkezésre.

PRODUKTORG
SZERVEZÉSI VÁLLALAT
Dél-Alföldi Iroda
H-6722 Szeged, Bokor u. 15. II. 2.
MNB 209-11249 1

COMMODORE 64

BASIC FELHASZNÁLÓI KÉZIKÖNYV

KÉSZÍTETTE: DR. ÚRY LÁSZLÓ
LEKTORÁLTA: ÚRY ÁGNES

LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
BUDAPEST, 1984

**„A MIKROSZÁMÍTÓGÉPEK ÉS ALKALMAZÁSI RENDSZEREIK
KUTATÁSA–FEJLESZTÉSE”**

című

OMFB TÁRCAPROGRAM

4/b. ALPROGRAMJA KERETÉBEN KÉSZÜLT

PROGRAMOZÓI SEGÉDLET

**OMIKK
LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
Telefon: 570–433/482**

Témafelelős: SZIKLAI KLÁRA

**ISBN 963 592 344 9
963 592 345 7**

Utánnyomás

**Készült
az OMIKK házi nyomdájában
Felelős vezető: Tóth Károly**

**Eng. sz.: 48179
T. sz.: 12843**

TARTALOMJEGYZÉK

I. kötet

1. fejezet

<u>Bevezetés</u> /18 oldal/	1.1
1.1 A kézikönyv feladata	1.1
1.2 Hogyan használjuk a könyvet?	1.2
1.3 Előismeretek	1.3
1.4 A C-64 mikroszámítógép üzembeállítása	1.3
1.5 A C-64 kiegészítő berendezéseinek kiválasztása	1.4
1.6 Bevezetés a mikroszámítógépek programozásába	1.7

2. fejezet

<u>Szövegszerkesztő</u> /6 oldal/	2.1
-----------------------------------	-----

3. fejezet

<u>BASIC interpreter</u> /28 oldal/	3.1
3.1 BASIC programok tárolása	3.1
3.2 BASIC változók, tárolásuk	3.5
3.3 Szintaxis	3.12
3.4 A BASIC interpreter működése	3.21

4. fejezet

<u>BASIC utasítások</u> /121 oldal/	4.1
4.1 BASIC alapszavak ABC sorrendben	4.3
4.2 SIMONS' BASIC utasítások	4.110

5. fejezet

Lemezegység /30 oldal/

5.1	Bevezetés	5.1
5.2	Az adatok tárolása	5.4
5.3	A 15. csatorna használata	5.8
5.4	Program file-ok használata	5.11
5.5	Szekvenciális file-ok használata	5.15
5.6	Véletlen file-ok használata	5.17
5.7	Gépi kódu programok	5.25
5.8	Relativ file-ok használata	5.28
	Utószó a második kiadáshoz	

II. kötet

6. fejezet

További perifériális egységek /31 oldal/

6.1	Bevezetés	6.1
6.2	Kazettás egység	6.2
6.3	Billentyűzet	6.6
6.4	Nyomtatás	6.8
6.5	RS-232-es csatorna	6.11
6.6	A játék I/O felhasználási lehetőségei	6.16
	A Commodore CP/M használata	6.23
6.7	A CIA chip működése	6.24
6.8	SIMONS' BASIC: I/O utasítások	6.30

7. fejezet

Grafikus lehetőségek /81 oldal/

7.1	Bevezetés	7.1
7.2	Karakter üzemmódok	7.10
7.3	Bit térképes üzemmód	7.26
7.4	A képernyő görgetése	7.31
7.5	További grafikus lehetőségek	7.33
7.6	Sprite-ok	7.36
7.7	SUPERGRAFIK	7.53
7.8	SIMONS' BASIC: grafikus utasításkészlet	7.65

8. fejezet

Hang generálás /29 oldal/

8.1	Bevezetés	8.1
8.2	A SID lehetőségeinek ismertetése	8.4
8.3	A burkológörbe /ADSR/ generátor	8.10
8.4	A SID chip további lehetőségei	8.16
8.5	Példák	8.22
8.6	SIMONS' BASIC SID utasításai	8.27

9. fejezet

Gépi kódu programozás /72 oldal/

9.1	A 6510 mikroprocesszor hardver lehetőségei	9.1
9.2	A 6510 processzor utasításkészlete	9.7
9.3	A Commodore-64 memóriaszervezése	9.40
9.4	KERNAL alprogramok	9.42
9.5	Monitorok, assemblerek, fordítók	9.57
9.6	HELP+	9.58

10. fejezet

A C-64 programozása /14 oldal/

10.1	Bevezetés	10.1
10.2	A BASIC nyelv használata	10.1
10.3	Gépi kódu programrészek felhasználása	10.8
10.4	BASIC beszúrások	10.13

Függelék /59 oldal/

HIBAÜZENETEK	F.1
BASIC KULCSSZAVAK	F.4
BASIC UTASÍTÁSOK - Összefoglalás	F.6
Standard BASIC programok átalakítása a Commodore-64 BASIC-re	F.11

Képernyő kódok	F.12
ASCII kódok	F.16
Képernyő és szinmemória	F.16
DOS hibaüzenetek	F.18
SIMONS' BASIC hibaüzenetek	F.21
Matematikai függvények	F.23
I/O eszközök csatlakozói	F.24
6510 processzor specifikációja	F.32
6526 CIA chip specifikációja	F.38
6566/6567 VIC II chip specifikációja	F.45
6581 SID chip specifikációja	F.51

1. Fejezet

BEVEZETÉS

1.1 A kézikönyv feladata

A könyv, amit az Olvasó a kezében tart kettős céllal készült. Alapvető feladata, hogy programozói kézikönyvként, a C-64-gyel végzett napi munkát segítse; a felmerülő problémákra - vagy legalább is a legtöbbjükre - választ adjon. Másik feladata, hogy segítséget nyújtson azoknak is, akik ennek segítségével szeretnék a C-64-en való programozás fogásait elsajátítani. A két cél nem könnyen egyeztethető össze. Egy mikrogép programozásának elsajátítása a ROM-ban tárolt program /jelen esetben a BASIC/ használatától halad a gépi-kódu /vagy ami a gyakorlatban ugyanaz az assembly/ programozás felé. A gyakorlott programozó programjainak jelentős része gépi kódu program. Éppen ezért a könyv - azon kívül, hogy külön fejezet foglalkozik az M6510 mikroprocesszor lehetőségeivel - igen sok utalást tartalmaz a különböző feladatok gépi-kódu megvalósítására is. Ezek a részek a kezdők számára nehezebbek, első olvasásra célszerű átugorni őket. A tankönyvként való használatot azzal is segítettük, hogy igen sok - és főleg egyszerű - BASIC és gépi kódu programrészt iktattunk a szövegbe. Ezek részletes áttanulmányozása, kipróbálása segít az utasítások jobb megértésében. A C-64-gyel és a programozással még csak most ismerkedőket segíti a könyvhöz külön megvásárolható lemez, amelyik számos mintapéldát tartalmaz.

Végül - a fent említett két célon kívül - volt egy harmadik, nehezen kifejezhető szempontunk is a könyv megírása közben. Ez a szempont szorosan kapcsolódik a BASIC programozási nyelv problémáihoz. A BASIC - tul azon, hogy igen hamar megtanulható - nem a világ legtokéletesebb programozási nyelve. Alapvető hibája, hogy nem strukturált, és ezért - a priori - nem támogatja az

algoritmusok, feladatmegoldások áttekinthető, könnyen módosítható, javítható, egyszóval strukturált leírását. Másik probléma a BASIC mikrogépeken való megvalósításával kapcsolatos, amelyek csak interpretert tartalmaznak, s ez igen lelassítja a program futását. Ehhez kapcsolódhat még az is, hogy a gép egyes lehetőségei BASIC-ből nehezen vagy egyáltalán nem érhetőek el. Operációs rendszerrel működő mikrogépek ezt a problémát természetesen megoldják.

Az első probléma - módszertani jellegénél fogva - nem tartozik szorosan témánkhoz. A második problémával kapcsolatban az az álláspontunk, hogy a BASIC gépi kódú rutinok felhasználásával igen jól használható bonyolult feladatok megoldására is. Erről legjobban a "Gépi kódú programozás" és a "BASIC programozási fogások" fejezetek olvasása győzheti meg az Olvasót, bár a többi fejezetben is érezhető ez a szoros kapcsolat.

1.2 Hogyan használjuk a könyvet ?

A könyv fejezetei többé-kevésbé függetlenek egymástól. Kezdőknek először a 2., 3. és 10. fejezetek olvasását ajánljuk, ezek azok, amelyek elsősorban a BASIC-kel foglalkoznak. Természetesen minden olyan esetben, amikor erre szükség van fel kell látni a 4. fejezetet, amelyik ABC sorrendben tartalmazza a BASIC kulcsszavakat és hatásukat. További fejezetek és alfejezetek vannak, amelyek a gépi kódú programozás ismerete nélkül is használhatók, ilyenek elsősorban a különböző perifériás egységekkel foglalkozó részek. A C-64 további lehetőségeinek kihasználásához azonban a gépi-kódú programozás ismeretére van szükség. A 9. fejezet ismerete - amelyik az M6510 processzor leírását tartalmazza - nagymértékben megkönnyíti a 4.-8. fejezetek gépi kódú utalásainak megértését.

Végül megjegyezzük, hogy a könyv elsősorban azért került kiadásra, mert a gyártók által szállított kézikönyvek nem tartalmaznak teljes és pontos információt a gépek hardver és szoftver rendszeréről. A könyv anyaga elsősorban nem a C-64-hez szállított leírásokból, hanem más könyvek, folyóiratok ötletei, leírásai - és a szerző saját tapasztalatai - nyomán állt össze.

1.3 Előismeretek

A könyv olvasásához a programozásban való bizonyos jártasság szükséges. Feltételezi a 2-es és 16-os számrendszer; valamint olyan alapvető fogalmak, mint értékadás, vezérlésátadás, ciklus, változó, adattípusok stb. ismeretét. Ezeket egy kezdő BASIC programozó már néhány hét alatt minden nehézség nélkül elsajátíthatja. Elsősorban az I/O-val kapcsolatos részek a hardver mélyebb ismeretét igénylik. A gépi-kódu programozásról szóló fejezet bevezető része lehetővé teszi, hogy a témával most először ismerkedők is megértsek.

1.4 A C-64 mikroszámítógép üzembeállítása

A C-64 mikroszámítógép hátoldalán illetve szemből nézve - a jobb oldalán számos csatlakozási lehetőséget találunk /lásd az 1.8 oldalt/ A kiegészítő berendezések megfelelő csatlakoztatása a helyes működésük alapfeltétele. A csatlakozók ugyan, úgy vannak kialakítva, hogy mindegyik csak saját ellendarabjával csatlakoztatható, de azért röviden összefoglaljuk a C-64 üzembeállításának legfontosabb lépéseit.

1. A C-64 központi egysége a billentyűzettel egybeépült, így azt nem kell külön csatlakoztatnunk. A C-64 transzformátora külön egység, ennek egyik vége a hálózati villásdugó. A másik kábelt nyomjuk be a C-64 jobb oldalán található hálózati csatlakozóba. A transzformátort magát úgy helyezzük el, hogy mind az - esetleg használt - lemezegységtől, mind a televíziós képernyőtől a lehető legmesszebb legyen.

2. A transzformátor villásdugóját csatlakoztassuk egy 220 V, 50 Hz-es hálózatba.
3. A C-64 központi egységhez szállított video-kábel segítségével a C-64 TV kimenetét és a televíziós képernyő antenna csatlakozóját kössük össze. Régebbi televíziókészülékek esetén antenna adaptere is szükség lehet. Ez külön kell beszerezni.
4. Helyezzük áram alá a televíziós készüléket, majd a C-64 számítógépet is.
5. Ezután kerülhet sor a televíziós készülék hangolására. Célszerű valamelyik VHF csatornát egyszer s mindenkorra a C-64 frekvenciájára hangolni. A televíziós készüléket úgy kell hangolni, hogy tisztán jelenjen meg az alábbi üzenet:

****** COMMODORE 64 BASIC V2 ******

64K RAM SYSTEM 38911 BASIC BYTES FREE

Szines készüléken a felirat világoskék keretben, sötétkék alapon kell, hogy megjelenjen.

6. Amennyiben további kiegészítő berendezéseket is csatlakoztatunk a C-64-hez, azt a C-64 bekapcsolása előtt kell elvégezni. Ebben az esetben először a kiegészítő berendezéseket helyezzük áram alá, csak legutoljára a C-64-et.
7. Egyes esetekben - például a C-64 modemén keresztül más számítógéphez történő kötése esetén - célszerű szakember segítségét is igénybe venni.

1.5 A C-64 kiegészítő berendezéseinek kiválasztása

A C-64 számítógépet tervezői több-célú, univerzális mikroszámítógépnek tervezték. Ennek megfelelően - kiépítettségétől függően - más és más feladatok megoldására alkalmas.

1/ Otthonon felhasználás /home-computer/ Elsősorban a VIC-II chip segítségével, a C-64-re kiváló játékprogramok írhatók. A játékok általában a billentyűzet segítségével is használhatók, de lényegesen kényelmesebb, ha botkormányokat és/vagy potmétereket használunk. A játékprogramok vagy ROM-ban vagy mágneses adathordozón szerezhetők be. Ennek megfelelően az otthoni /játék, demonstráció, szórakozás/ célokat kielégítő legkisebb konfiguráció a következő:

- a/ C-64 központi egység
- b/ színes televíziós készülék
- c/ botkormányok és/vagy potméterek
- d/ játékprogramok
 - bővítő egységek /cartridge/ égetve vagy
 - mágneses adathordozón /ekkor egy kazettás vagy lemezegység is szükséges/.

2/ Programfejlesztés Felhasználói programok elkészítéséhez már komolyabb hardver és szoftver feltételekre van szükség. A legkisebb konfiguráció, amelyik ezt lehetővé teszi a következő:

- a/ C-64 központi egység,
- b/ VDU /nem feltétlenül színes!/,
- c/ lemezegység /pl. VIC 1541/,
- d/ nyomtató,
- e/ szoftver.

A C-64 BASIC interpreter nem igazán alkalmas felhasználói programok fejlesztésére. Egy C-64 programfejlesztői környezetnek legalább az alábbi szoftver elemeket kell tartalmaznia:

- i/ gépi kódu monitor,
- ii/ assembler,
- iii/ a BASIC valamilyen kiterjesztése, vagy valamilyen más programozási nyelv, például PASCAL, FORTH stb.,
- iv/ nagy adattömböket kezelő programok esetén egy DOS /lemezegység/ monitor,

v/ AMT feladatok megoldásához a grafikus lehetőségeket támogató programok.

Ezek a szoftver-elemek általában beszerezhetők. Külön is kiemelnénk a CP/M kártyát, amelyik lehetővé teszi a Z-80 bázisu programok felhasználását.

3/ Adatfeldolgozás A C-64 adatelőkészítő, feldolgozó mikrogép-ként csak akkor használható, ha legalább 2 lemezegység tartozik hozzá. Ennek két oka is van, egyrészt a soros buszon az adatáramlás meglehetősen lassu, másrészt egy minifloppira kevés / ~170 K/ adat fér csak el. A minimális konfiguráció ennek megfelelően a következő:

- a/ C-64 központi egység,
- b/ VDU /nem feltétlenül színes!/,
- c/ legalább 2 lemezegység /pl. VIC 1541/,
- d/ nyomtató /speciális, pl. bizonylat nyomtatók!/,
- f/ felhasználói szoftver.

4/ Intelligens terminál A C-64 kiképzése lehetővé teszi, hogy akár intelligens terminálként egy nagyobb géphez, vagy önálló gépként egy hálózathoz csatlakoztathassuk. Ebben az esetben már további hardver eszközökre is szükség van, amelyik lehetővé teszi az összeköttetés létrehozását. Amennyiben lényegesen nagyobb teljesítményű számítógépekkel kötjük össze a C-64-et egy lemezegység általában már nem elég a gép üzemeltetéséhez. Ennek megfelelően az általunk ilyen esetre ajánlott minimális konfiguráció a következő:

- a/ C-64 központi egység,
- b/ VDU /nem feltétlenül színes!/,
- c/ lemezegység /de inkább 2!/,
- d/ RS-232 kártya /a bővítő egység csatlakozójába/,
- e/ modem,
- f/ hálózati szoftver.

A modem megválasztása a kialakítandó kapcsolat típusától függ, és kiválasztása feltétlenül szakember bevonását igényli.

A C-64 BASIC interpreter önállóan elvégzi az RS 232-höz kapcsolódó feladatokat. Ez azonban nem mindig elegendő /lokális hálózatok kialakításához például nem kell az RS-232/. Másrészt a kommunikáció lebonyolításához, adatkonverzióhoz feltétlenül további szoftverre van szükség. /Bizonyos géptípusokra ez már beszerezhető./

1.6 Bevezetés a mikroszámítógépek programozásába

Az első fejezet ezen utolsó paragrafusa azok számára készült, akik eddig még semmilyen jártasságra sem tettek szert a mikroszámítógépek programozása és használata terén. A paragrafus megértéséhez a C-64 használatára van szükség, ezért a következő konfiguráció meglétét feltételezzük:

- a/ C-64 központi egység,
- b/ televíziós képernyő,
- c/ lemezegység,
- d/ a könyvhöz külön megvásárolható oktató lemez.

A c/ és d/ pontokban felsoroltakra nem mindig lesz szükség.

Az utasítások bevitele

A C-64 számítógép üzembehelyezése /lásd 1.3/ után bejelentkezik, majd kiírja a

READY.

Üzenetet. A READY alatt egy villogó téglalap, az ugynevezett kurzor látható. A READY. /KÉSZ/ üzenet azt jelenti, hogy a számítógép befejezte tevékenységét és további parancsra vár. /A bekapcsolás után ez a tevékenység a számítógép inicializálása volt./

A kurzor azt a helyet mutatja, ahová a következő, általunk begépelte karakter kerül. Ha lenyomjuk például a ? billentyűt,

akkor a kurzor helyén megjelenik egy kérdőjel, és a kurzor egy hellyel jobbra lép. Ha már nincs hely a sorban, akkor a kurzor a következő sor elejére kerül. Az utasítást - bármi legyen is az - karakterenként kell bevinnnünk /nem úgy mint például a ZX81 esetén./ Egy speciális billentyű, a [RETURN] szolgál annak jelzésére, hogy befejeztük a parancs gépelését. Ezután a BASIC interpreter végrehajtja az utasítást, majd a következő parancs begépelésére vár.

Ha például a ?2+2 begépelése után megnyomjuk a [RETURN] billentyűt, válaszként a következőt kapjuk:

4

READY



A legegyszerűbb esetben a READY. üzeneten kívül semmi egyéb üzenetet nem kapunk. Ez azt jelenti, hogy parancsunkat a gép hibátlanul végre tudta hajtani. Gépeljük be például a következőt: ?2/Ø [RETURN] . A képernyőre a következők kerülnek:

?DIVISION BY ZERO ERROR

READY



A READY üzenet kiírása előtt egy további - ugynevezett - hibaüzenetet kaptunk, amelyik jelzi, hogy milyen típusu probléma miatt nem volt képes az interpreter a parancsot végrehajtani. Jelen esetben ez pusztán annyit jelent, hogy Ø-val kíséreltünk meg osztani. Az interpreter hibaüzeneteit az egyéb üzeneteitől az különbözteti meg, hogy kérdőjellel /?/ kezdődnek.

A C-64 lehetőséget biztosít, hogy a képernyő tetszőleges helyén látható információt parancsként használjuk fel. Így például a fenti, hibás ?2/Ø sort a következőképpen javíthatjuk ki. A [CRSR←] billentyűt annyiszor nyomjuk le, míg a kurzor nem kerül a

fenti hibás sorba. Ezután a [CRSR→] billentyű háromszori lenyomásával elérhetjük, hogy a kurzor a Ø fölé kerüljön. Ha most megnyomjuk az [5] billentyűt a Ø eltűnik és helyette az 5 jelenik meg. Ezután megnyomva a [RETURN] billentyűt a következő választ kapjuk:

.4

DIVISION BY ZERO ERROR

READY.

■ villog!

/A READY nem törli a képernyőt, ezért az előző üzenet még látszik, de nincs hiba!/
/

A szövegek begépelésére - írógépen használt terminológiával élve - három váltó is szolgál.

Váltók használata nélkül az a karakter kerül a képernyőre, amelyik a billentyűn látható, vagy amennyiben két jel van a billentyűn, akkor azok közül az alsó.

A [SHIFT] váltót használva /ami azt jelenti, hogy a SHIFT lenyomva tartása közben lenyomunk egy billentyűt/ a következőket kapjuk. Olyan billentyűk esetén, amelyekre két jel van ráírva a siftelés /emelés/ a felső jelet eredményezi, a neki megfelelő karakter kerül beírásra. Ha a billentyűn egyetlen jel látható, akkor a siftelés a billentyű gombján elől látható jobboldali karakter begépelését eredményezi.

A [C=] váltók használatával a billentyűk gombján elől és baloldalt látható karaktereket lehet begépelni.

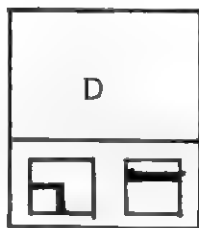
A [CTRL] és [C=] váltóknak még néhány speciális feladata is van, amelyekről később még szólnunk.



A C-64 a televíziós képernyőn két karakterkészletet tud megjeleníteni /de nem egyidőben/. Az egyiket 'nagy betűk/grafikus jelek' karakterkészletnek hívjuk. Ha ezt használjuk a betűket csak nagybetűs alakjukban tudjuk a képernyőre írni. A másikat 'kis betűk/nagy betűk' karakterkészletnek hívjuk, mert ebben az esetben a betűk kis betűkként íródnak a képernyőre, míg a [SHIFT] váltót használva nagybetűk kerülnek a képernyőre /ez felel meg az írógépnek/.

Egyik karakterkészletről a másikra a [SHIFT] és a [C=] váltók egyidejű megnyomásával térhetünk át.

A váltók hatását az egyes karakterkészletek használata esetén a következőkben foglalhatjuk össze:

nagy betűk/grafikus jelek




D = [D]
 = [SHIFT-D]
 = [C=-D]



3 = [3]
 # = [SHIFT-3]

kis betűk/nagy betűk

d = [D]
 D = [SHIFT-D]
 = [C=-D]

3 = [3]
 # = [SHIFT- 3]

Bizonyos billentyűk megnyomásának nem az a hatása, hogy valamely karakter megjelenik a képernyőn. Ilyenek voltak például a [CRSR↑] illetve a [CRSR→] billentyűk, vagy a váltók. Összefoglaló néven ezeket vezérlő karaktereknek hívjuk. A 2. fejezet részletesen összefoglalja ezek hatását.

Szinvezérlés A szinvezérlést segítő billentyűkről külön is szólnunk. A CTRL lenyomva tartása közben egy szinbillentyű leütése az eredményezi, hogy azt követően valamennyi karakter olyan szinnel íródik ki a képernyőre. A C= váltó és a szinbillentyű használata további 8 szint ad. Ezeket részletesen a Függelékben találhatjuk meg /F.15 oldalon/.

Kurzor vezérlés A kurzor billentyűk teszik lehetővé, hogy a kurzort - a képernyőn látható információ megváltoztatása nélkül - tetszőleges helyre pozicionáljuk.

További vezérlő karakterek szabályozzák az inverz alakban való kijelzést, a képernyő törlését stb.

A C-64, mint zsebszámológép Legegyszerűbb használata a C-64-nek az, amikor műveletek elvégzésére, bizonyos értékeknek a memóriában való tárolására használjuk. /Ezek azok a feladatok, amelyeket egy zsebszámítógéppel el lehet végezni/. Ehhez mindössze két BASIC utasítást kell ismerni, ezek az

a/ értékadó

b/ nyomtató

utasítások.

Az értékadó utasítások $\langle \text{változó} \rangle = \langle \text{kifejezés} \rangle$ alaknak, ahol a $\langle \text{változó} \rangle$ az a név, amelyben a kifejezés értékét tárolni szeretnénk. A legegyszerűbb értékadások azok, amelyek a $\langle \text{változó} \rangle$ értékét konstansként adják meg:

$X = 2.53$ vagy $Y = -15.001$

Ennél egy fokkal bonyolultabb a $Z = X * X + Y * Y$ értékadás. A változók és kifejezések írásának meghatározott szabályai vannak, és ez lényegében a matematikában tanult szabályokkal egyezik meg. Részletesen erről a 3. Fejezetben szólnunk.

A nyomtató utasítás szerkezete, használata az értékadó utasításénál lényegesen bonyolultabb. Itt csak egyik legegyszerűbb alakját adjuk meg, ami céljainknak teljes egészében megfelel. /Bővebben a 4. Fejezetben írjuk le a használatát/.

```
PRINT <kifejezés1 > , <kifejezés2 > , ...
```

A PRINT helyett egyszerűen egy kérdőjelet is írhatunk.

Most már mindent tudunk, hogy a C-64-gyel bonyolult számításokat is végezhesünk. Nézzük meg, hogyan számíthatjuk ki egy derékszögű háromszög átfogóját /C/, ha adott a két befogó /A és B/. Legyen A=4, B=3.

A következő parancsokat kell végrehajtani:

```
A=4 [RETURN]
B=3 [RETURN]
? SQR (A↑2 + B↑2) [RETURN]
```

Az utolsó sort a következővel is helyettesíthetjük:

```
C = SQR (A↑2 + B↑2) [RETURN]
? C [RETURN]
```

Ebben az esetben előbb kiszámítjuk a C értékét, tároljuk és csak utána írjuk ki.

A C-64 - a számokon túl - **szövegekkel is képes dolgozni**. Tetszőleges karaktersorozatot **sztringnek hívunk**. Azok a változók, amelyek sztringeket tárolnak **\$-ra végződnek**. Például a

```
X$="KUTYA" : Y$ = "FULE" [RETURN]
?X$+Y$ RETURN
```

parancsok végrehajtása után a képernyőn a KUTYAFULE felirat jelenik meg. A kettőspont **/:/** az első sorban két parancs egymás után

írására szolgál, így a [RETURN] billentyűt csak a legvégén kell benyomni. Sztringek körében egyetlen 'hagyományos' művelet van csak, az összeadás, ami a sztringek egymáshoz fűzését jelenti.

Eredeti példánkat így is módosíthatjuk:

```
A=4:B=3:C=SQR(A^2 + B^2) [RETURN]
?"BEFOGOK=";A,B: ? "ATFOGO="; C [RETURN]
```

Programok betöltése és futtatása

A C-64 zsebszámológépként való használatán kívül a másik legegyszerűbb eset, amikor mások által megírt programot akarunk használni. Ehhez természetesen valamilyen háttértárnak, például lemezegységnek a rendelkezésünkre kell állnia. Vegyük elő az oktatólemez és töltsük be az első programot! /Ez nem más, mint az oktatólemez használatát leíró program/.

A programot a

```
LOAD "*",8 RETURN
```

parancs segítségével tölthetjük be. Ennek hatására a lemezegységen levő indikátor kigyullad, jelezve, hogy a C-64 és a lemezegység közt adatforgalom zajlik. A töltés végrehajtása után a READY üzenet megjelenik a képernyőn. A

```
RUN [RETURN]
```

parancs végrehajtása után a képernyőn megjelenik a használati utasítás első oldala. A program futását a [STOP] billentyű megnyomásával bármikor megszakíthatjuk. Vannak olyan programok is, amelyek futása csak a C-64 kikapcsolásával szakítható meg.

Programok írása

A BASIC interpreter lehetőséget biztosít arra is, hogy a begépelte parancsokat ne hajtsa végre azonnal, hanem tárolja a memóriájában, ahol később végrehajtható lesz. Ahhoz, hogy egy parancs programsorrrá váljon nem kell mást tenni, mint eléje egy sorszámot

irni. Ha egy sor számmal kezdődik, az mindig programsornak számít.

A derékszögű háromszög példáját programként a következőképpen lehet megírni:

```
10 A=4:B=3:C=SQR(A↑2 + B↑2) [RETURN]
20 ?"A BEFOGOK=";A,B [RETURN]
30 ?"AZ ATFOGO=";C [RETURN]
```

Ezután a

```
RUN [RETURN]
```

parancs kiadása a fenti programot lefuttatja és ugyanazt az eredményt adja, mintha a megfelelő parancsokat mi magunk hajtottuk volna végre.

Problémát jelent azonban, hogy a fenti programmal csupán a 3, 4 befogóju derékszögű háromszög átfogóját képes kiszámítani. Ahhoz, hogy tetszőleges derékszögű háromszög átfogóját kiszámítsa egy új utasítást kell használni, amivel egy már futó programnak adatokat adhatunk át. Az új program 10. sora így alakul:

```
10 INPUT A,B : C = SQR(A↑2 + B↑2) RETURN
```

A RUN [RETURN] billentyűzése után a képernyő következő sorának elején egy kérdőjel /?/ jelenik meg; jelezve, hogy a program adatot vár /ez az INPUT utasítás feladata/. Az adatok billentyűzése után a megfelelő értékadások végrehajtódnak; majd a program fut tovább. A

```
3,4 [RETURN]
```

válasz után az INPUT A,B utasítás hatására A értéke 3, B értéke 4 lesz és a program tovább fut. Ilyen módon természetesen tetszőleges derékszögű háromszög átfogóját kiszámíthatjuk.

Vezérlésátadó utasítások

Normális körülmények közt a BASIC interpreter a memóriájában tárolt programsorokat sorszámaik növekvő sorrendjében hajtja végre. A fenti programban az utasítások végrehajtásának sorrendje: 10, 20, 30. Vannak azonban olyan utasítások, amelyek feladata éppen az utasítások végrehajtási sorrendjének megváltoztatása. Ezek közül sorolunk fel néhányat. Részletes leírásuk a 4. Fejezetben található meg.

a/ GOTO <sorszám>

Az utasítás hatására a program futása a <sorszám> -nak megfelelő programsortól folytatódik. Ha több derékszögű háromszög átfogóját is ki akarjuk számítani, akkor a fenti programot célszerű a következő sorral kiegészíteni:

```
40 GOTO 10 [RETURN]
```

A program futása így újra és újra visszatér a program elejére és tetszőleges számú derékszögű háromszög átfogóját ki tudjuk számítani.

b/ IF <feltétel> THEN GOTO <sorszám>

Az utasítás végrehajtása a GOTO-ban megjelölt sorszámú utasítással folytatódik, ha a <feltétel> igaz. Ha nem, a program a következő utasítás végrehajtását kezdi el. Előző programunkat kiegészíthetjük az adatok valódiságát ellenőrző sorral:

```
15 IF A <= 0 OR B <= 0 THEN GOTO 10
```

A fenti feltétel szokásos alakja $(B \leq 0 \text{ vagy } C \leq 0)$. Ha ez teljesül, akkor rossz adatokat adtunk meg és a program a végeredmény kiírása helyett újra kéri az adatokat.

c/ FOR <valós változó> = <kezdőérték> TO <végérték>

Ez az utasítás az ugynevezett ciklus-utasítás, mert segítségével lehetővé válik bizonyos programrészek megadott számú ismét-

lése. Ha például előre tudjuk, hogy N darab derékszögű háromszög átfogóját kell kiszámítani, akkor ezt a következő programmal végezhetjük el:

```
5 INPUT N: FOR I=1 TO N
10 INPUT A,B: C=SQR(A^2+B^2)
15 IF A<=0 OR B<=0 THEN GOTO 10
20 PRINT "BEFOGOK=";A,B: PRINT "ATFOGOK=";C
25 NEXT I
```

READY.

/Ne felejtsük el a programsorok végén a [RETURN] billentyű lenyomását!/
 /

A RUN parancs kiadása után először az N értékét kell megadnunk /pl 3./ Ezt követően 3 derékszögű háromszög átfogóját számíthatjuk ki. I értéke a kezdőérték vagyis 1 lesz és a program a következő NEXT utasításig fut. Ezután I értéke megnő 1-gyel és ellenőrzi az interpreter, nem léptük-e túl a 3-at. Ha nem, akkor a 10-20 közti programsorok újból végrehajthatódnak. Ha igen, a következő utasítás kerül végrehajtásra. /Ha ilyen nincs, a program futása véget ér/.

Szerkesztő utasítások

A szövegszerkesztő utasításaival már megismerkedtünk. További utasítások a memóriában tárolt program szerkesztését /módosítását/ teszik lehetővé

a/ LIST

A fenti utasítás a képernyőre listázza a memóriában tárolt programot. Ha csak a program bizonyos sorszámú soraira vagyunk kíváncsiak, akkor meg kell adnunk az első, illetve utolsó listázandó sort. Például LIST 5 az 5. sorszámú sort listázza ki./

b/ A kilistázott programsorokat a szövegszerkesztő vezérlő karaktereivel / [CRSR↑], [DEL] stb./ szerkeszthetjük. A kívánt módosítások után a [RETURN] megnyomása a memóriában tárolt programsort a képernyőn láthatóra cseréli.

c/ A program egy adott sorát egy ugyanolyan sorszámú üres sor bevitelével törölhetjük. Például billentyűzzük be a következőt:

```
15 [RETURN]
```

Ennek hatására törlődik a 15. sor /ha volt ilyen sorszámú utasítás/. Nincs mód több programsor egyidejű törlésére.

d/ NEW

A parancs hatására törlődik a memóriában tárolt program. Új program írása előtt célszerű a parancs kiadása.

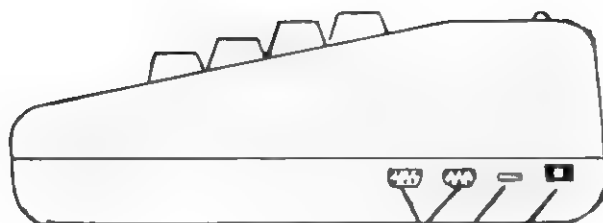
1.7 Az oktatólemez használata

A C-64-re tervezett, külön megvásárolható oktatólemez kettős feladatot lát el. Egyrészt kellően dokumentált példaprogramokat tartalmaz, amelyek lehetővé teszik a C-64 sajátosságainak megismerését; amelyek továbbfejlesztve beépíthetők felhasználói programokba. Másrészt egy BASIC bővítést tartalmaz, amelyik biztosítja, hogy a gép használata közben a C-64-re vonatkozó információk a képernyőre kérhetők. Például a !CHR\$ /uj/ parancs hatására a képernyőre kerül a CHR\$ BASIC utasítás ismertetése. A #CHR\$ /uj/ parancs viszont a CHR\$ kódokat listázza a képernyőre.

Az oktatólemez használati utasítása magán a lemezen található. Mint már említettük a

```
LOAD "*" ,8 RETURN
```

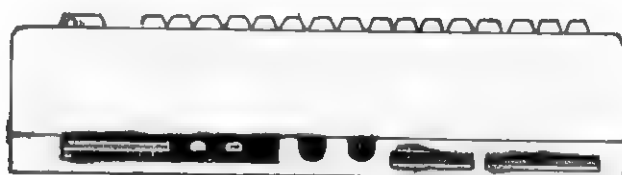
parancs betölti azt a programot, amelyik ismerteti az oktatólemez felhasználási lehetőségeit. A lemezen levő programokat úgy terveztük, hogy bármilyen könyvi segédlet nélkül, önmagukban is használhatóak legyenek.



Játék I/O csatlakozók

Hálózati kapcsoló

Hálózati csatlakozó



Bővítő egység csatlakozója

TV csatorna választó

TV csatlakozó

AUDIO/VIDEO csatlakozó

Soros kimenet

Kazettás egység csatlakozó

Felhasználói kapu csatlakozója

A C-64 csatlakozói

2. Fejezet

SZÖVEGSZERKESZTŐ

A C-64 számítógép inicializálása, a READY kiírása, vagy az INPUT végrehajtása közben a vezérlés a szövegszerkesztő rutinnak adódik át, melynek segítségével a teljes képernyőt használhatjuk a szükséges információ bevitelére. A [RETURN] billentyű megnyomása után a szövegszerkesztő azt feltételezi, hogy a kurzort tartalmazó sort kívántuk bevinni, és azt adja át feldolgozásra az interpreternek.

Ha például egy program kódszövegét kilistáztuk a képernyőre, akkor a kurzor billentyű és más speciális billentyűk használatával mozoghatunk a képernyőn, így - a megfelelő helyre érve - bármilyen változtatást elvégezhetünk. Miután befejeztük az összes kívánt javítást a szöveg megadott számú sorában, és lenyomtuk a [RETURN] billentyűt (bárhol a sorban), akkor ennek hatására a szövegszerkesztő elolvassa a teljes 80 karakteres logikai képernyő sort. A szöveg ezután kerül az interpreterhez feldolgozásra és tárolásra. A javított sor a sor régi változatának a helyére kerül a memóriában. Ha kulcsszó-rövidítést használt, aminek eredményeként a programsor a listázáskor meghaladja a 80 karaktert, a 80-on felüli karaktereket elveszíti a sor javításakor, mert a szerkesztő csak két fizikai képernyő sort olvas. Ez az oka annak is, hogy az INPUT utasítás használata több mint 80 karakterre nem lehetséges. Így, gyakorlati megfontolásokból, a listázható BASIC szöveg sorának hossza 80 karakter lehet, abban a formában, ahogyan a képernyőn megjelenik. Bizonyos feltételek mellett a képernyő szerkesztő a kurzor vezérlő billentyűt szabályszerű üzemmódtól eltérően kezeli. Ha a kurzor páratlan számú idézőjel /"/ után következik, akkor a szerkesztő az úgynevezett "idézőjel" üzemmódban működik.

"Idézőjel" üzemmódban a 'közönséges' karakterek szabályszerűen tárolódnak, de a kurzor vezérlő karakterek a továbbiakban nem mozgatják a kurzort, ehelyett inverz karakterek íródnak ki, amelyek valójában a lenyomott kurzor vezérlőt jelentik. Ugyanez igaz a szinkiválasztó billentyűkre is. Ez lehetővé teszi, hogy a programokba sztringek közé kurzor és színvezérlő karaktereket tegyünk. Ennek köszönhető, hogy amikor az idézőjelben levő szöveg kiíródik a képernyőre, akkor a sztring részeként automatikusan végrehajtja a kurzor elhelyezést és a szín-vezérlő funkciót. Példa, sztringben levő kurzor vezérlő használatára /gépélje be a következő sort/:

```
10 PRINT "A[R][R] B [L][L][L] C [R][R] D": REM R=[CRSR JOBB] L=[CRSR BAL]
```








A számítógép ezt írja ki:

```
AC BD
```

A [DEL] billentyű az egyetlen kurzor vezérlő, amelyre az "idézőjel" üzemmód nincs hatással. Így, ha egy hibát csinál miközben "idézőjel" üzemmódban gépel, a [← CRSR] billentyű nem használható visszaléptetésre és átirásra - még akkor sem, ha az

[INST] billentyű inverz video karaktereket hoz létre. Ehelyett fejezze be a teljes megkezdett sort, és ezután a [RETURN] billentyű lenyomását követően a szokott módon javítsa ki a hibát. Egy másik lehetőség, /ha már nincs szüksége kurzor vezérlőre a sztringben/, az, hogy lenyomja a [RUN/STOP] és a [RESTORE] billentyűket, amelyek megszüntetik az "idézőjel" üzemmódot. A sztringben használható kurzor vezérlő billentyűket az alábbi táblázatban mutatjuk be:

Kurzor vezérlő karakterek "idézőjel" üzemmódban

<u>Vezérlő billentyű</u>		<u>Megjelenési forma</u>
kurzor fel	[CRSR ↑]	
kurzor le	[CRSR ↓]	
kurzor balra	[CRSR ←]	
kurzor jobbra	[CRSR →]	
CLR	[CLR/]	
HOME	[/HOME]	
INST	[INST/]	

Ha nem "idézőjel" üzemmódot használunk, a [SHIFT] billentyű lenyomva tartása és ezután az [INST] billentyű lenyomása a kurzortól jobbra levő karaktert egy hellyel jobbra tolja és így egy szóközt hoz létre a két karakter között, ahová új karaktert írhatunk be. A szerkesztő ezután 'inzert' üzemmódban kezd működni addig, amíg az összes így megnyitott szóköz be nem töltődik.


Inzert üzemmódban a kurzor és a szín-vezérlő karakterek ismét inverz karakterként jelennek meg. Az egyetlen különbség az [INST/DEL] inzert-törlő billentyűn van. A [DEL] ahelyett, hogy szabályosan működne, mint "idézőjel" üzemmódban, most inverz 'T' -t hoz létre. A [INST] billentyű, amely inverz karaktert hoz létre "idézőjel" üzemmódban, szabályosan szóközt szur be. Ez azt jelenti, hogy olyan PRINT utasítás hozható létre, amelyik törlést /DEL/ tartalmaz, ami nem lehetséges idézőjel üzemmódban. Az inzert üzemmódot a [RETURN] , [SHIFT] és [RETURN] , vagy a [RUN/STOP] és a [RESTORE] billentyűk lenyomásával lehet törölni. Ezen kívül befejeződik az inzert üzemmód akkor is, ha betöltjük az összes közbeszurt karakter-

helyet. Példa[DEL]karakterek használatára sztringben:

```
10 PRINT "HELLO" [DEL] [INST] [INST] [DEL] [DEL] P "
```

Amikor ezt a példa-programot lefuttatjuk, a HELP szó fog kiíródni, mert az LO betűk törlődnek, mielőtt a P kiíródna. A sztringben levő törlő karakter LIST és PRINT utasítással egyaránt működik. Ezzel a módszerrel el lehet rejteni a sor egy részét vagy a szöveg egy teljes sorát. Az ilyen sorok szerkesztése természetesen igen nehézkes.

Az eddig említetteken kívül is van még néhány olyan karakter, amely speciális módon írható csak ki. Ahhoz, hogy hozzáférjünk ezekhez, részükre üres helyeket kell hagyni a sorban, lenyomni a [RETURN] billentyűt, és utána javítani a sort. Ekkor nyomjuk le a [CTRL] /vezérlő/ billentyűt és a [RVS/ON] billentyűt, hogy elkezdhessük az inverz karakterek gépelését. Az inverz alaku vezérlő karakterek a következők:

<u>Funkció</u>	<u>Billentyűzés</u>	<u>Megjelenési forma</u>
siftelt RETURN	[SHIFT] M	
kis/nagy betűkre kapcsolás	N	
nagy betűkre/grafikákra kapcsolás	[SHIFT] N	

A[SHIFT] billentyű lenyomva tartása és a [RETURN] billentyű leütésének hatására a képernyőn egy kocsi vissza és egy sorrelés történik, de nem fejeződik be a sztring. Ez a LIST és a PRINT utasítással egyaránt működik, így a javítás majdnem lehetetlen, ha ezt a karaktert használjuk. Ha az elsődleges output a CMD utasítás hatására a nyomtatóra megy, az inverz "N" karakter a nyomtatót kis/nagy betű karakter készletre, a

[SHIFT N] a nyomtatót 'nagy betűk/ grafikai jelek' karakter készletre váltja át.

Inverz karaktereket a [CTRL] billentyű lenyomva tartásával és a [RVS ON] lenyomásával írhatunk be "idézőjel" üzemmódban, amelynek hatására egy inverz R jelenik meg az idézőjelen belül. Az ezután beírt karakterek a nyomtatás során inverz formában kerülnek kiírásra. Az inverz kiírás hatásának megszüntetésére a [CTRL] billentyű lenyomva tartása közben nyomjuk meg a [RVS OFF] billentyűt. /Ennek hatására egy inverz grafikus jel kerül a sztringbe.

```
10 PRINT "[ CTRL-RVS ON] KUTYAFULE [CTRL - RVS OFF] "
20 PRINT " KUTYAFULE"
```

program kétszer írja ki a "KUTYAFULE" sztringet, először inverz formában, utána normálisan.

A PRINT utasításban szereplő vezérlő karaktereket legegyszerűbben CHR\$(B) alakban állíthatjuk elő. Az alábbi táblázat összefoglalja a felhasználható vezérlő karaktereket:

<u>Funkció</u>	<u>CHR\$(B)</u>
kis betűk/nagy betűk karakterkészlet	14
nagy betűk/grafikák karakterkészlet	142
[SHIFT - C=] letiltása	8
[SHIF - C=] engedélyezése	9
Home	19
CLR	147
Kurzor le	17
Kurzor fel	145
Kurzor jobbra	29
Kurzor balra	157

C-64

2.6

Inverz karakterek	18
Normál karakterek	146
Törlés	20
Beszurás	148
Fehér	5
Piros	28
Zöld	30
Kék	31
Fekete	144
Bibor	156
Sárga	158
Cián	159
Narancs	129
Barna	149
Világos piros	150
Szürke 1	151
Szürke 2	152
Világos zöld	153
Világos kék	154
Szürke 3	155

3. Fejezet

BASIC INTERPRETER3.1 BASIC programok tárolása

A BASIC interpreter két üzemmódban működik: az első az úgynevezett parancs /kalkulátor vagy végrehajtási/ mód, a másik a program /tanulási, tárolási/ mód. Az, hogy melyik üzemmódban működik, az az utoljára bevitt sor első nem szóköz karakterétől függ. A [RETURN] megnyomása után az éppen bevitt sor átmásolódik egy 80 karakter hosszú pufferbe. A puffer a 2. lapon található /\$0200-\$0253/. Ezután kerül sor a puffer első nem szóköz karakterének tesztelésére. Ha ez szám, akkor a BASIC úgy tekinti, hogy egy programsort vittünk be és elhelyezi a memóriába a többi sor közé. Ha az első karakter nem számjegy, akkor parancsnak értelmezi és végrehajtja. A puffer feldolgozása; a pufferben vagy a programterületen tárolt programok végrehajtása ugyanugy történik programsorok illetve parancsok esetén, ezért most csak a programsorokkal foglalkozunk.

A tárolt programsorok a LIST utasítás segítségével ellenőrizhetők, de így semmilyen információt sem kapunk arról, hogyan helyezkednek el a memóriában. Megpróbálhatjuk byte-onként kiírni a programot a következő egysoros paranccsal:

```
X=0: FOR I= 2049TO 2249:POKE 1024+X, PEEK(I):X=X+1:NEXT
```

A fenti program a memória 2049-es címétől kezdve 200 byte-ot átmásol az 1024-es címtől kezdődően. Általában 2048-nál /\$0800/ kezdődik a tárolt BASIC program és 1024-nél /\$0400/ a képernyő memóriája. A fenti program tehát a képernyő tetejére kiírja a program első 200 byte-jának megfelelő karaktereket. Az

eredmény ilyen lesz:

```
@RHE@ 4,4:K2048@-HF@RV$$(13):ROF$$(
146)@FHJ@ I 5: J 4@ HT@ 1000
:K31: J: 4, (13);@-H1@ I : @ HCY
(K)@HCR@HLC Y32 Y Y364: 1070@
HFD Y364 1070@HFD Y366 1070@
```

Jobb eredményre számíthatunk, ha a byte-ok értékét, nem pedig a nekik megfelelő karaktereket listázzuk ki. A későbbiek jobb megértése kedvéért az értékeket hexadecimálisan iratjuk ki. Erre szolgál a következő BASIC-rutin, amelyik a memória tartalmát írja ki, egy adott helytől kezdve:

```
10 OPEN 3,4:CMD3
20 V=2048 :REM TXTTAB BEALLITASA C-64-RE
30 K=4:X=V:GOSUB 100:REM CIM KIIRASA
40 PRINT"M"+L$+" ";:K=2
50 FOR I=1 TO 8 :REM 8 BYTE KIIRASA
60 X=PEEK(V):V=V+1
70 GOSUB 100:PRINTL$+" ";
80 NEXT I
90 PRINT:PRINT:GOTO 30
100 REM X HEXADECIMALIS ALAKJA
110 REM L$-BA KERUL. L$ HOSSZA K
120 L=X:L$=""
130 FOR J=1 TO K
140 LZ=L-INT(L/16)*16
150 L=INT(L/16)
160 L$=CHR$(48+LZ-(LZ>9)*7)+L$
170 NEXT J
180 RETURN
```

Hogyan tudunk ezzel a BASIC programmal egy másik BASIC programot olvasni ? /Hiszen ha új programként írjuk be, akkor a vizsgálni kívánt program eleje elvész!/ Egyik lehetséges megoldás, hogy a fenti programot a vizsgálni kívánt program után írjuk be. /A program sorszámai ekkor természetesen megváltoznak/. A másik - ennél érdekesebb mód a következő parancs kiadása:

```
POKE 43, PEEK(45) : POKE 44, PEEK (46) : LOAD "MEM", 8
```

A tárolt BASIC program első byte-jának helyét (43) jelöli, míg (45) a program szövege utáni első helyre mutat. A lemezen a MEM nevű file tartalmazza a fenti programot. A két POKE utasítás után a BASIC interpreter a BASIC elejének az új értéket tekinti, s oda tölti be a programot. A program futtatása, ha a 10 ? "HOGY VAGY?" program beírása után töltöttük be a következőt eredményezi:

M0300	00	15	08	0A	00	99	20	22
M0808	48	4F	47	59	20	56	41	47
M0810	59	20	3F	22	00	1B	08	14
M0813	00	80	00	00	00	48	14	5C
M0820	01	0A	20	02	05	01	04	04

A következőkben részletesen ismertetjük, hogyan tárolja a BASIC interpreter a programsorokat. A sorszámot 2- byte-os egész számként tárolja. Ennek értékének a $0 \leq x < 55280$ / $\Rightarrow \$FF00$ / intervallumba kell esnie. /Ha valamilyen programhiba során egy sorszám felső byte-ja \$FF-re változik, a rendszer azt nem-létezőként kezeli!/. Szokás szerint először az alsó, majd a felső byte kerül tárolásra. Ezt követően a szövegben előforduló BASIC

gének jelzésére és a mutató relativ; a következő sor elejének az aktuális sor elejétől mért távolságát mutatja. Ez maximum 255 lehet. /Igy programsoronként egy byte-ot megta-
karithatunk/. A program tárolási módjának ismerete számos esetben hasznos lehet. Egyrészt jobban megérthetünk bizonyos eljárásokat, másrészt könnyen megvalósíthatunk olyan BASIC funkciókat, amelyek a C-64 BASIC-ben nincsenek implementálva. Például egy meggondolatlan NEW hatását néhány POKE-kal semlegesíthetjük. Részletes példákat a 10. fejezetben adunk.

3.2 BASIC változók, tárolásuk

A 'változó' fogalma az algebrából került át a számítástechnikába. Eredetileg egy pontosan meg nem nevezett mennyiség jelölésére szolgált. Ebben az értelemben mi nem használjuk. Algebrai értelemben $X=X+1$ lehetetlen. A korai BASIC interpreterek ezért tették kötelezővé a LET használatát az értékadó utasításban: LET $X=X+1$ /pl.ZX81-en/. Az Algol erre az $X:=X+1$ jelölést használja. A számítástechnikában éppen ezért célszerűbb a változókat tároló egységeknek felfogni. Ekkor az $X=X+1$ értékadás jobb oldalán szereplő X a tároló tartalmát jelenti az utasítás végrehajtása előtt, a baloldali X pedig a tároló tartalmát jelenti a végrehajtás után. A Microsoft BASIC interpreterek a változók három típusát különböztetik meg, attól függően, hogy egész, valós vagy sztring mennyiségek tárolására szolgál. Ezen kívül megkülönböztetnek egyszerű vagy indexes /tömb/ változókat. Az interpreter a változók típusát a nevet követő \$ illetve % jelből állapítja meg. Ezek jelentik a sztring illetve egész változókat. Ilyen jel hiányában az interpreter a változót valós típusnak tekinti.

A valós és egész típusok közti konverziót a gép automatikusan végzi. Például az $L\%=L/256$ értékadásban $L/256$ -ot egészre kerekíti, ellenőrzi, hogy a $[-32768, 32767]$ intervallum-

128	\$80	END	167	\$A7	THEN
129	\$81	FOR	168	\$A8	NOT
130	\$82	NEXT	169	\$A9	STEP
131	\$83	DATA	170	\$AA	+
132	\$84	INPUT#	171	\$AB	-
133	\$85	INPUT	172	\$AC	*
134	\$86	DIM	173	\$AD	/
135	\$87	READ	174	\$AE	↑
136	\$88	LET	175	\$AF	AND
137	\$89	GOTO	176	\$B0	OR
138	\$8A	RUN	177	\$B1	>
139	\$8B	IF	178	\$B2	=
140	\$8C	RESTORE	179	\$B3	<
141	\$8D	GOSUB	180	\$B4	SQN
142	\$8E	RETURN	181	\$B5	INT
143	\$8F	REM	182	\$B6	ABS
144	\$90	STOP	183	\$B7	USR
145	\$91	ON	184	\$B8	FRE
146	\$92	WAIT	185	\$B9	POS
147	\$93	LOAD	186	\$BA	SQR
148	\$94	SAVE	187	\$BB	RND
149	\$95	VERIFY	188	\$BC	LOG
150	\$96	DEF	189	\$BD	EXP
151	\$97	POKE	190	\$BE	COS
152	\$98	PRINT#	191	\$BF	SIN
153	\$99	PRINT	192	\$C0	TAN
154	\$9A	CONT	193	\$C1	ATN
155	\$9B	LIST	194	\$C2	PEEK
156	\$9C	CLR	195	\$C3	LEN
157	\$9D	CMD	196	\$C4	STR\$
158	\$9E	SYS	197	\$C5	VAL
159	\$9F	OPEN	198	\$C6	ASC
160	\$A0	CLOSE	199	\$C7	CHR\$
161	\$A1	GET	200	\$C8	LEFT\$
162	\$A2	NEW	201	\$C9	RIGHT\$
163	\$A3	TAB(202	\$CA	MID\$
164	\$A4	TO	203	\$CB	GO
165	\$A5	FN			
166	\$A6	SFCC			

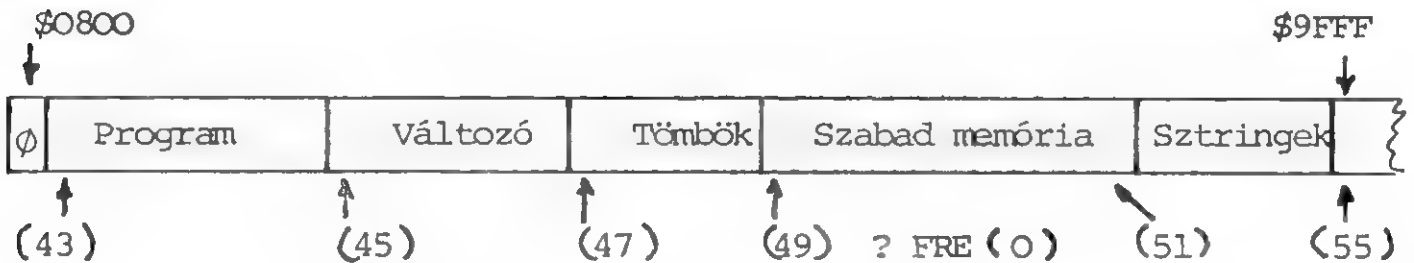
ba esik-e s ha igen L%-hez hozzárendeli ezt az új értéket. Sztringek és számok közti konverzióra külön beépített függvényeket használhatunk: $L\% = \text{STR}\%(L)$, $L = \text{VAL}(L\%)$, $L\% = \text{VAL}(L\%)$. Két további konvertáló függvény a $\text{CHR}\%$ és az ASC , amelyek azonban csak egy-egy byte-on működnek.

A változó nevek képzésének szabályai a következők:

- (i) Az első karakter csak betű lehet, (A-Z).
- (ii) A következő karakter tetszőleges alfanumerikus karakter (0-9, A-Z) lehet,
- (iii) Ezt tetszőleges számú alfanumerikus karakter követheti, de ezek nem képezik a név részét. (KATA és KATI tehát ugyanaz a név).
- (iv) Ezt esetleg egy \$ vagy % követheti, jelezve, hogy a változó sztring, illetve egész típusú.
- (v) A következő karakter egy (jel lehet, jelezve, hogy tömbváltozóról van szó. Ezt követik az indexek vesszővel elválasztva, majd egy) jel. Az indexek és a (jel nem részei a névnek.
- (vi) A név nem tartalmazhat egyetlen BASIC alapszót sem /Például VALI, WORD nem megengedett nevek/. A védett változók, TI, ST, TI\$ lehetnek változó nevek részei mert nem számitanak alapszónak (ATTILA megengedett és egyenlő AT-val) .

Változók, akár parancs módban, akár program futása közben használjuk őket a memóriában tárolt program után következnek. Kivételt csak a sztringek képeznek. A fenti módon csak a nevük és egy mutató kerül tárolásra, amelyik a sztring első karakterére mutat. Sztring-műveletek végrehajtásakor mindig külön meg kell győződnie az interpreternek arról, van-e elég hely a memóriában. Ha nincs, a program futása ?OUT OF MEMORY ERROR hibaüzenettel félbeszakad.

A BASIC program, illetve a változók tehát így helyezkednek el a memóriában:



A tárolt program egyes komponenseinek első byte- mutatnak a \emptyset .
lapon tárolt mutatók:

BASIC program kezdete	(43) (\$2B)	általában $\$0801$
BASIC terület vége	(55) (\$37)	általában $\$9FFF$
BASIC változók kezdete	(45) (\$2D)	
BASIC tömbök kezdete	(47) (\$2F)	
BASIC tömbök vége + 1	(49) (\$31)	
BASIC sztringek eleje	(51) (\$33)	

A BASIC munkaterületet ($\$0800-\$9FFF$) a (43), (55) mutatókkal magunk is állíthatjuk. A többi mutató értéke a program szerkesztésétől illetve futásától függően alakul.

Egyszerű változók tárolása. Tetszőleges, nem-tömb változó 7 byte-nyi helyet foglal el a memóriában. Ezen túlmenően a sztring változók - a hosszuknak megfelelően - további byte-okat foglalnak el a BASIC munkaterület végén. Kivételt képeznek a programban szereplő szövegkonstansok. Ezek esetében a sztring-változó mutatója a memória megfelelő helyére mutat. A tömbök az egyszerű változók után helyezkednek el,

így például az XT változó első használata azt eredményezi, hogy a tömböket 7 byte-tal feljebb tolja az interpreter, majd az XT változót helyére rakja. A tárolás nem a leg-tömörebb, egész és sztring változóknál 3 illetve 2 felesle-ges byte is szerepel.

Tömbváltozók tárolása. A tömbváltozók tárolása a tömbre vonatkozó legfontosabb adatok tárolásával /név, dimenzió-szám, egyes dimenziók nagysága/ kezdődik, majd ezt követik az egyes tömbelemeknek megfelelő értékek tárolása 5, 3 il-letve 2 byte-on, attól függően, hogy valós, sztring vagy egész típusu tömbről van-e szó. A tömbváltozó tartalmaz még egy relatív mutatót, amelyik a következő tömbváltozó első byte-jára mutat. A tömbelemek oszlopfolytonosan helyezked-nek el a memóriában, a DIM A (1,2) deklaráció hatására például ilyen sorrendet kapunk:

$A(\emptyset, \emptyset)$, $A(1, \emptyset)$, $A(\emptyset, 1)$, $A(1, 1)$, $A(\emptyset, 2)$, $A(1, 2)$.

A C-64 a változókat a következőképpen tárolja:

Típus:

Név:

Tartalom:

lebegőpontos

ASCII	ASCII v. \emptyset	exponens	mantissza			
			M1	M2	M3	M4

előjel bit

egész

ASC+128	ASC+128 v. 128	FELSŐ	ALSÓ	\emptyset	\emptyset	\emptyset
---------	-------------------	-------	------	-------------	-------------	-------------

sztring

ASCII	ASC+128 v. 128	Hossz	Mutató		\emptyset	\emptyset
			Alsó	Felső		

függvény
definíció

ASCII + 128	ASCII v. \emptyset	Mutató a defini- cion	Mutató a vál- tozon		\emptyset	
		Alsó	Felső	Alsó	Felső	

A név két byte-on való tárolását az teszi lehetővé, hogy az alfanumerikus jelek ASCII kódjai mind 128-nál kisebbek. A nevet tartalmazó byte-ok 7-ik bitje így felhasználható a fenti 4 típus jelzésére.

Tömbváltozók

Tömbnév	Rel.mutató		Dim. szám	Utolsó dim+1		..	Első DIM+1		..ADATOK
	ALSÓ	FELSŐ		FELSŐ	ALSÓ		FELSŐ	ALSÓ	

Egy tömb a memoriában

$$5+2 * \text{DIMENZIÓSZÁM} + (\text{DIM}_1+1) * \dots * (\text{DIM}_N+1) * 2, 3 \text{ vagy } 5$$

helyet foglal el / 2 = egész, 3 = sztring, 5 = valós típusu/.

Sztring tömbváltozó esetén ehhez még az egyes tömbelemek sztringjeit is hozzá kell számítani, hiszen a tömb maga csak a sztringek hosszát és a mutatókat tartalmazza. Ha például a DIM X\$ (1000) definíciót használjuk és minden egyes X\$ (N) sztring 10 hosszú, akkor az X\$ tömb összesen

$$5+2+1001 * 3 + 1001 * 10 = 13020 \text{ byte.}$$

Az A% (30,10) tömb /amelyik 31 * 11 egész számot tartalmaz
 ≈ 300 / helyfoglalása

$$5+2 * 2 + 31 * 11 * 2 = 696 \text{ byte.}$$

Végül két BASIC programozási példát adunk, amelyek segítségével a fenti BASIC mutatók, illetve a program változóinak tárolása jól vizsgálható. Az első programrészt bármely program után be lehet vinni, majd a program megállása helyett

átadni erre a rutinra a vezérlést.

Az alább megjelölt sorokban a módosításokra azért volt szükség, hogy a kezdeti, ne pedig ezzel a rutinnal bővített program adatait kapjuk.

```
PRINT"3" : K=425
DEF FN DEEK(X)=PEEK(X)+256*PEEK(X+1)
PRINT"PROGRAM ELEJE:";TAB(15);FN DEEK(43)
PRINT"VALTOZOK ELEJE:";TAB(15);FN DEEK(45)-K
PRINT"PROGRAM HOSSZA:";TAB(15);FN DEEK(45)-FN DEEK(43)-K;"BYTE"
PRINT
PRINT"VALTOZOK SZAMA:";TAB(15);(FN DEEK(47)-FN DEEK(45))/7-3
PRINT
PRINT"TOMBOK ELEJE:";TAB(15);FN DEEK(47)-K-21
PRINT"TOMBOK VEGE:";TAB(15);FN DEEK(49)-K-21
PRINT
PRINT"SZTRINGEK ELEJE:";TAB(15);FN DEEK(51)
PRINT"SZTRINGEK VEGE:";TAB(15);FN DEEK(55)
```

A következő program arra szolgál, hogy a képernyőn láthatóvá tegyük, hogy futás közben, hogyan helyezi el az egyes változókat a memóriában. E célból a "BASIC változók kezdete" mutatót a képernyő-memória második sorának elejére, a "BASIC munkaterület vége" mutatót pedig a képernyő-memória utolsó helyére állítjuk. A program futtatása lehetőséget biztosít, hogy a képernyő első sorát felhasználva különböző értékeket vigyünk be és figyeljük, hogyan tárolja a gép ezeket az értékeket. A program futtatása után könnyen meglehet, hogy semmit sem látunk a képernyőn. Ennek oka, hogy a változók tárolása olyan értékeket helyez a memóriába, amelyeknek nem felelnek meg látható jelek. A program megállása

C-64

3.12

után vigyünk a kurzort a képernyő megfelelő része fölé, a tárolt jelek láthatóvá válnak.

```

10 PRINT"J"
20 POKE 45,40:POKE 46,4
30 POKE 55,217:POKE 56,7 :CLR
40 J=13*4096+8*256 :REM MINDEN JEL LATHATO!
50 FOR K=J TO J+999: POKE K,14 :NEXT
60 A=1234:FOR I=1 TO 500:NEXT I
70 DIM SZ$(10)
80 FOR I=0 TO 10
90 INPUT"SZTRING=";SZ$(I):NEXT I:END

```

3.3 Szintaxis

A BASIC nyelvet gyakran szokás "angol" nyelvünek is nevezni. Ez a kifejezés teljesen félrevezető, mert igaz ugyan, hogy a BASIC kulcsszavak angol szavak is egyben, de a programok "nyelvtani szerkezetének" nem sok köze van az angol nyelvhez. A BASIC parancsok, sorok írását éppen úgy meg kell tanulnunk, mint bármely más program nyelvét. De még ebben az esetben is lehetnek értelmezési problémák. Helyesnek tekinthetünk-e egy 10 GOSUB 132 utasítást, ha a 132-es sor nem létezik?! Hogyan fogalmazzuk meg a DATA...RESTORE...READ utasítások egymáshoz való viszonyát? A problémát legegyszerűbben úgy hidalhatjuk át, ha az egyes BASIC komponenseknek külön-külön definiáljuk a szintaxisát. Ebben az esetben a parancs végrehajtása közben legalábbis ?SYNTAX ERROR hibajelzést nem kapunk.

C-64

3.13

A következőkben röviden összefoglaljuk az alapvető szintaktikus szabályokat, amelyek betartása - hacsak nem használunk gépi kódu rutinokat - kötelező. A könyv függelékében a C-64 BASIC szintaxisát Bacchus-Naur formában is definiáljuk.

3.3.1. Konstansok A C-64 BASIC három fajta konstans használatát engedi meg. Ezek azonosak a három típussal:

- 1/ egész;
- 2/ valós;
- 3/ sztring

konstansok.

1/ Az egész konstansok előjeles egész számok. A + előjel kiírása nem kötelező. A számnak a $[- 32768, 32767]$ zárt intervallumba kell esnie, ugyanis a konstans használatakor az interpreter 2- byte-os egész szám alakra konvertálja át az egész konstansokat. Programsorok írásakor nem kerül sor konverzióra. A -1234 konstansot az interpreter 5 byte-on tárolja.

2/ Valós konstansok a következő részekből állhatnak:

- (i) előjeles egész szám (I) ;
- (ii) tizedes pont (.) ;
- (iii) előjel nélküli egész szám (F) ;
- (iv) az exponens jele (E) ;
- (v) előjeles egész szám (K) .

A fentiek közül a . pont vagy az E jel használata kötelező. Ha ezek hiányoznak, a gép a fenti konstanst egésznek tekint. Az (i) - (v) részekből álló valós konstans értéke,

C-64

3.14

leszámítva a kerekítési hibákat:

$$\left(1 + \frac{F}{10^n}\right) \cdot 10^K,$$

ahol n az F törtrész jegyeinek számát jelenti. Az
 (i) - (iii) részek alkotják tehát a mantisszát, a
 (iv) - (v) részek pedig a kitevőt.

A legnagyobb, illetve legkisebb pozitív valós szám, amit az
 interpreter még tárolni tud:

1.70141183E + 38 illetve. 2.93873588E - 39

Ha valamilyen számítás eredményeként ennél nagyobb számot
 kapunk a végrehajtás ?OVERFLOW ERROR hibaüzenettel megsza-
 kad. Ha a számítás eredménye a fenti számnál kisebb, a vég-
 rehajtás folytatódik. Az interpreter a 0.0 értékkel számol
 tovább.

A valós konstansok alakjának ismerete igen lényeges a nyom-
 tatási kép megtervezéséhez. A PRINT utasítás a valós tipu-
 su változók értékeit 9 jegyre kerekítve nyomtatja ki.
 Amennyiben a szám nem kisebb .01-nél és nem nagyobb 999999999-nál
 a kiírás nem tartalmaz exponenst. Ha a fenti feltétel nem
 teljesül, a nyomtatási kép mantisszája mindig 1.-nél kisebb,
 de 0.1-nél nagyobb szám lesz, s a megfelelő exponens is kii-
 ródik. Erről részletesen szólnunk a PRINT utasítás leírásánál.

Példák valós konstansokra

2.43	-. 99937	+3.1926	235.2E6
235E 6	- . 2E - 13	.1E - 2	.0123

3/ Sztring /vagy szöveg/ konstansok alfanumerikus és grafikus jelek sorozatát jelentik. A sztring konstansok hosszát csak az köti meg, hogy a képernyőről legfeljebb 80 karakter hosszú sort lehet bevenni. A sztring konstansokat idézőjelek `"/` közé kell zárni. Így az egyetlen alfanumerikus jel, ami nem szerepelhet sztring konstansban, az maga az idézőjel. Sztring változó értékeként előálló sztringben már szerepelhet, a `CHR$ (34)` felhasználásával.

Példák sztring-konstansokra

`"HOGY VAGY?"`

`"25.000.000$ ELEG"`

`" "`

`" " /üres sztring, nem egyenlő CHR$ (0) -val !`

3.3.2. Változók. A változók /egyszerű, illetve tömbváltozók/ képzésének szabályairól már volt szó az előző paragrafusban. Egy parancs vagy program végrehajtása során az interpreter először ellenőrzi, hogy megfelel-e a képzési szabályoknak, majd megnézi, hogy a memóriában szerepel-e, ez a változó. Ha nem, akkor a 3.2-ben ismertetett módon elhelyezi, és beállítja a kezdő értékét. Ez számváltozók esetén 0 illetve 0.0, sztring esetén pedig a `"/üres/` sztring. Tömbváltozók esetén ellenőrzi a dimeziószámát, az egyes indexek értékeit is. Amikor egy tömb első elemére először hivatkozunk, akkor a tömb valamennyi eleme bekerül a memóriába.

3.3.3. Kifejezések. Az algebrában tanultakhoz hasonlóan konstansokból, változókból és egy- illetve többváltozós műveletekből építhetünk fel kifejezéseket. A kifejezéseknek két fajtája van:

1/ aritmetikai;

2/ sztring

kifejezősek.

Először a sztring kifejezésekről szólunk, mert ez az egyszerűbb eset. Sztring kifejezések a következők:

- (i) sztring változók;
- (ii) két sztring kifejezés a +jellel összekapcsolva;
- (iii) minden olyan függvény, amelynek a nevében a \$ jel szerepel. Ezek: CHR\$, LEFT\$, MID\$, RIGHT\$, STR\$. /Természetesen a függvények argumentumainak megfelelő típusu kifejezéseknek kell lenniük./
- (iv) a sztring kifejezések tetszés szerint zárójelezhetők.

Példák sztring kifejezésekre

```
"AB CD EF"      G$ + " N "      CHR$( N )
STR$(25) + RIGHT$(K$, 3, 6 )
"NEW " + A$ + B$.
```

A sztring-függvények hatásáról a következő fejezetben részletesen lesz szó. A + sztring-művelet a sztringek egymáshoz fűzését, kompozícióját jelenti. Például

```
"KUTYA" + "FUL" + "E" = "KUTYAFULE".
"ABRAKA" + "DABRA" = "ABRAKADABRA".
```

Az "" /üres/ illetve CHR\$(0) sztringek hatása a + művelet esetén nem ugyanaz:

```
A$ + "" = A$,  A$ + CHR$( 0 ) ≠ A$
```

bármilyen is volt az A\$ sztring értéke. Ugyanakkor
LEN("") = 0, LEN(CHR\$(0)) = 1 !

Megjegyezzük, hogy a DEF FN utasítás segítségével további sztring függvényeket nem tudunk definiálni .

Az aritmetikai kifejezések esete nem csak azért bonyolultabb, mert több művelet végezhető velük, hanem mert az aritmetikai kifejezések, mint speciális esetet magukban foglalják a logikai kifejezéseket. A logikai értékeket az interpreter 2-byte-os egész számként tárolja, s a megfelelő logikai műveletet /és, nem, vagy stb./ valamennyi biten egyszerre elvégzi. Ilyen módon logikai művelet egész változók közt is végezhetünk /például `X% AND B%` értelmes/. Az interpreter a /2-byte-os egész számként tárolt/ 0-t hamisnak, az ettől eltérő értéket igaznak tekinti.

Az összehasonlítás eredményeképpen azonban mindig 0-t vagy -1-et kapunk /-1 = `$FFFF`; az igaz érték tehát mind a 16 bitet 1-re állítja !/.

Relációs jelek A relációs jelek (`<`, `<=`, `>=`, `>`, `<>`, `=`) segítségével két számot, vagy két sztringet hasonlíthatunk össze. Ha a számok típusa eltérő, az egész számot lebegőpontosra alakítja át és utána hajtja végre az összehasonlítást. Ha a két mennyiség eleget tesz a relációnak eredményül -1-et, ha nem 0-t kapunk.

A relációs jelek értelme, amikor számokat hasonlítunk össze a szokásos:

<code>=</code>	egyenlő
<code><></code>	nem egyenlő
<code>></code>	nagyobb
<code><</code>	kisebb
<code>>=</code>	nagyobb vagy egyenlő
<code><=</code>	kisebb vagy egyenlő

Sztringek esetén `=` és `<>` jelentése ugyanaz. `A$ < B$` jelentése a következő. Sorba vesszük az `A$`, `B$` sztringekben szereplő jeleket, és megnózzuk melyik az a hely, ahol először

eltérnek. Ha ilyen nincs, akkor $A\$ < B\$$ pontosan akkor teljesül, ha $A\$$ rövidebb, mint $B\$$. Ha ilyen hely van, akkor $A\$ < B\$$ azt jelenti, hogy az első eltérő helyen szereplő jel ASCII kódszáma az $A\$$ sztringben kisebb, mint $B\$$ -ban.

Példák összehasonlításra

$1 = 4$	hamis (0)
$14 > = 62$	hamis (0)
$14 < 62$	igaz (- 1)
$5 * 5 < > 16$	igaz (- 1)
" A " < " D "	igaz (- 1)
" A " < " 9 "	hamis (0)
"XYZ" > "XY"	igaz (- 1)
"XYZU" > = "XY9"	hamis (0)
" " < = CHR\$ (0)	igaz (- 1)
CHR\$ (0) < = "□"	igaz (-1)/bármilyen jel szerepel is a □ helyén./

Logikai műveletek A BASIC interpreter összesen három logikai műveletet ismer, egy egyváltozós /NOT/ és két kétváltozós /AND, OR/ műveletet. Hatásukról részletesen a 4. fejezetben szólnunk.

Aritmetikai műveletek A BASIC interpreter a négy alapműveletet, valamint a hatványozás műveletét ismeri. Ezek jele a szokásos:

+	összeadás
-	kivonás
*	szorzás
/	osztás
↑	hatványozás

Az interpreter ezenkívül felismeri az ellentett képzést is: $-X\%$ például az $X\%$ ellentetjét jelenti. A gyökvonás az $X \uparrow (1/Y)$ kifejezés segítségével végezhető el. Az interpreter az algebrából ismert precedencia szabály szerint dolgozik. Egyenrangu műveletek esetén szigorúan balról jobbra haladva végzi el a műveleteket. Például $A/B/C$ algebrai alakja $\frac{a}{b.c}$!! / A kerekítési hibák miatt az algebrából ismert azonosságok nem mindig teljesülnek! /

Aritmetikai függvények A BASIC interpreter a legfontosabb matematikai függvényeket beépített rutinként ki tudja számítani. Ezek a következők: ABS, ASC, ATN, COS, EXP, FRE, INT, LEN, LOG, PEEK, POS, RND, SGN, SIN, SQR, TAN, VAL. A felsorolt 17 egyváltozós függvény természetesen nem csak a matematikai függvényeket tartalmazza.

Megjegyezzük, hogy a DEF FN utasítás segítségével további aritmetikai függvényeket tudunk definiálni.

Ennyi előzetes után definiálhatjuk, hogyan is épülnek fel az aritmetikai kifejezések:

- (i) tetszőleges egész vagy valós változó, logikai kifejezés egyben aritmetikai kifejezés;
- (ii) ha aritmetikai kifejezéseket műveleti jelekkel összekapcsolunk, akkor újból aritmetikai kifejezést kapunk;
- (iii) egy aritmetikai függvény, utána zárójelben a megfelelő típusu argumentum ugyancsak aritmetikai kifejezés;
- (iv) két aritmetikai vagy sztring kifejezés összehasonlítása logikai kifejezést eredményez;
- (v) egész és/vagy logikai kifejezések a logikai műveletekkel összekötve újabb logikai kifejezést adnak;

(vi) aritmetikai kifejezések tetszés szerint zárójellezhetőek.

Példák aritmetikai kifejezésekre

```

A + B + . 23
C ↑ ( D + E ) / 2
((X-C) ↑ ( D - E ) / 2 ) * 10 ) + 1
K % = 1 AND M < > X
NOT ( D = E )
K % = 2 OR ( A = B AND M < X )
FN DEEK ( X ) + FN DEEK ( X+2 ) = 0.

```

A fenti definíció egy igen érdekes sajátosságára hívnánk fel a figyelmet. Aritmetikai kifejezések szerepeltetése a logikai kifejezések közt megszokott. A C-64 azonban megengedi logikai kifejezéseknek aritmetikai kifejezésekben való felhasználását is. Kis gyakorlással elérhető, hogy IF...THEN... a programban alig szerepeljen.

Tekintsük a következő értékadást:

```
L% = ASC ( L$ ) - 48 + ( ASC ( L$ ) > 64 ) * 7
```

L\$ egy hexadecimális számot tartalmaz karakteres alakban. A fenti értékadás L%-hoz a karakternek megfelelő számértéket rendeli.

Műveletek sorrendje Mindig a magasabb rangú műveletek kerülnek végrehajtásra. Egyenrangú műveletek esetén a műveletek végzése balról jobbra halad. A műveletek a következő sorrendben kerülnek végrehajtásra:

C-64

3.21

1.	\uparrow	hatványozás
2.	$-$	ellentett képzés
3.	$* /$	szorzás, osztás
4.	$+ -$	összeadás, kivonás
5.	$> = <$	összehasonlítás
6.	NOT	logikai nem
7.	AND	logikai és
8.	OR	logikai vagy

3.3.4. További megjegyzések

A BASIC interpreter a szintaxis definiálásában nem következetes. Tartalmaz például egy aritmetikai kifejezés kiértékelő rutint. Néhány byte átírásával, s ennek a rutinnak a felhasználásával elérhető lenne, hogy GOTO X% alaku utasítás is végrehajtható legyen, ne kelljen a GOTO után pozitív egész konstansnak állnia. Még nyernénk is az ügyön, hiszen az ON...GOTO... utasítást teljes egészében elhagyhatnánk.

A 4. fejezetben az egyes utasításoknál külön ismertetjük azok szintaxisát.

3.4 A BASIC interpreter működése

A C-64 szövegszerkesztő a [RETURN] billentyű benyomását minden esetben úgy értelmezi, hogy befejeztük a BASIC parancs vagy programsor bevitelét és átadja a vezérlést a

BASIC interpreternek (\$ A480) . Az interpreter első lépésként meghív egy rutint (\$ A560) , amelyik a billentyűzetről bevitt utolsó sort, /amely jelenleg a képernyő-memóriában található meg/ áttölti az input pufferbe. Erre azért van szükség, hogy a

? " [CLEAR] EREDMENYEK = "

típusú utasítások végrehajthatók legyenek. Ha az interpreter az utasításokat a képernyő-memóriából venné ki, a fenti utasítás a [CLEAR] végrehajtása után elveszne. /Ez a probléma természetesen nem merül fel abban az esetben, ha a képernyő egy meghatározott része - például az utolsó sor - szolgál a parancsok, programsorok bevitelére. Ilyen a ZX81 rendszere./ Az input puffer a 2. lapon helyezkedik el a \$0200-\$0258 címeken. A fenti rutin az inputsor végére egy 0 byte-ot is elhelyez.

A BASIC szövegek feldolgozásában igen fontos szerepet játszik a /\$7A/ mutató, amelyik a memória azon helyére mutat, ahol az éppen feldolgozás alatt álló byte található.

Ennek a byte-nak az akkumulátorba való töltésére szolgál a CHRGET rutin. /\$0073-\$008A/. Ha a rutinba a \$73 ponton lépünk be, először eggyel növeli a (\$7A) mutató értékét, majd addig növeli, míg egy szóköztől különböző byte-ot talál. Végül két ellenőrzést hajt végre. A C jelzőbitet törli, ha ASCII számjegyet talált, különben 1-re állítja. A Z jelzőbit csak abban az esetben 0, ha a byte nem nulla és nem kettőspont.

Magasra állítja tehát abban az esetben, ha az akkumulátor értéke # \$0 vagy # \$3A. A rutin \$0079 belépési pontja nem

C-64

3.23

növeli eggyel a mutató értékét. Ezt szokás CHRGOT rutin-
nak is hívni.

Miután az interpreter a bevitt sort elhelyezte az input puf-
ferbe a (\$7A) mutató értékét a puffer elejére, vagyis \$200-ra
állítja, majd meghívja a CHRGOT rutint. Ha ez számjegyet ta-
lált az első nem szököz helyen, akkor az inputsort program-
sornak értelmezi és elhelyezi a memóriában. Ha nem, akkor
parancsnak értelmezi és végrehajtja. Ez utóbbi jelzésére
(\$58) -ba \$FF kerül.

Nézzük először azt az esetet, amikor parancsot vittünk be.
Az interpreter ennek felismerése után meghívja a \$A57C alatt
található tokenizáló rutint. Ennek a rutinnak a feladata,
hogy az input pufferben található sorban megkeresse a BASIC
alapszavakat, ezeket token-jükkel helyettesítse. Ez a rutin
ismeri fel a kulcsszavak rövidítéseit, a ? utasítást. Ügyel
arra, hogy páros számú idézőjel után szabad csak az alapsza-
vakat tokenjükre cserélni.

A rutin hibája, hogy a REM utasítás utáni programrészt is to-
kenizálja, ami elég szerencsétlen eredménnyel járhat. Az in-
putsor kódolása általában a sor rövidülését eredményezi.

Ezután a (\$7A) mutató értékét visszaállítja az inputsor ele-
jére és a \$A7D8 belépési ponton keresztül belép abba az el-
lenőrző ciklusba, amelyik a tokenizált BASIC szövegeket vég-
rehajtja. Ez a belépési pont nem egyezik meg azzal, ahová a
RUN utasítás után belép a program. A ciklust a későbbiekben
elemezzük.

Ha az inputsor programsornak bizonyult, először a sorszám 2-
byte-os egész számmá való átkonvertálására kerül sor. Ha ez

C-64

3.24

hibát eredményez /tul nagy a szám/ az interpreter hibajelzést küld és visszatér a szerkesztőbe. Ha nem, sor kerül a maradék sor tokenizálására az előbb már említett rutin /belépési pont \$A57C/ segítségével. Az interpreter ezután ellenőrzi, szerepel-e ilyen sorszámú sor a programban. Ha igen, akkor ezt törli a memóriából és az utána szereplő sorokat lejjebb mozgatja a memóriában. Ezután megvizsgálja van-e elég hely a sor elhelyezésére. Ha nincs, a program végrehajtása ?OUT OF MEMORY ERROR hibaüzenettel megszakad és a vezérlés visszaadódik a szövegszerkesztőnek. Ha van hely, és az inputsor nem üres, a program megfelelő helyén annyival feljebb tolódik a memóriában, hogy a sor éppen beférjen. Ilyen módon természetesen a programsorok elején álló - és a következő sor elejére mutató - számok elromlanak. Ezért kerül sor egy további rutin /belépési pont \$A533/ végrehajtására, amelyik ujraszámítja ezeket a mutatókat.

Ha az inputsor üres /tehát egyedül a sor végét jelző 0 byte-ból áll/ az utolsó lépésekre nem kerül sor. Egy üres sor /pl. 213 **[RETURN]**/ bevitele így a sor törlését eredményezi.

Végül utoljára hagytuk a BASIC-kulcsszavak végrehajtását ellenőrző ciklus ismertetését. Két belépési pontja van. Az egyik, amikor parancsot hajtunk végre, ez a \$A708 pont. A másik belépési pont, amelyen keresztül a RUN parancs végrehajtása során lép be, a \$A7AE pont. A rutin ellenőrzi, nincs-e lenyomva a **[STOP]** billentyű, majd sor kerül a következő BASIC-byte olvasására. Megvizsgálja, hogy ez nem sorvégét jelző nulla-e. Direkt módban a nulla a ciklusból való kilépést, a szövegszerkesztőbe való visszatérést eredményezi. Program végrehajtása közben a nulla a következő sor elején álló mutató ellen-

C-64

3.25

őrzését jelenti. Ha ez ϕ , akkor END nélkül ért véget a program, a vezérlés visszatér a szövegszerkesztőbe. Ha nem, a sor-száma a \$3B-3C byte-okba töltődik, a (\$7A) mutatót pedig a BASIC programszöveg elejére állítja az interpreter.

Mindezek után kerül sor az alábbi rutin végrehajtására, amely egyben a prancs mód esetén az ellenőrző ciklus belépési pontja.

A rutin a CHRGOT rutin segítségével megvizsgálja a következő karaktert, amely

- | | |
|-----|-----------------|
| (a) | kettőspont; |
| (b) | ASCII karakter, |
| (c) | token; |
| (d) | egyéb |

lehet. Minden egyes esetben a rutin egy belépési pont címét helyezi a verembe. A belépési pont annak a tevékenységnek az elejére mutat amit az adott esetben végre kell hajtani. Az /a/ esetben nincs tennivaló. A /b/ esetben az interpreter LET utasítást tételez fel; s az ennek megfelelő belépési pontot tölti a verembe. A /c/ esetben a BASIC ROM elején levő táblázatból megkeresi az ahhoz a tokenhez tartozó belépési pontot és ezt tölti be a verembe. A /d/ esetben a ?SYNTAX ERROR üzenetet kinyomtató hibarutin kezdőcíme töltődik a verembe.

A fenti rutin végrehajtása után az interpreter egy újabb RTS utasítást hajt végre, ami a verembe töltött belépési pontra való ugrást eredményezi.

A belépési pontoknak megfelelő rutinok végén - hacsak nem történt hiba - a vezérlés az ellenőrző ciklus legelejére adó-

dik vissza - függetlenül attól, hogy direkt vagy program módban vagyunk-e.

Az egyes tokeneknek - illetve utasításoknak - megfelelő belépési pontokat, a hozzájuk tartozó rutin működésével együtt a 4. fejezetben részletesen ismertetjük. Most külön csak a RUN végrehajtásáról szólunk. A RUN utasítás végrehajtása /belépési pont \$A870/ a (\$7A) mutatónak a BASIC szöveg elejére való állításával kezdődik. Ezután a vezérlés az ellenőrzési ciklusba kerül az \$A7AE belépési ponton keresztül.

A BASIC interpreter fenti működésének ismerete elengedhetetlen a BASIC lehetőségeinek kiterjesztéséhez. A fent jelzett rutinok legtöbbje ugyanis egy JMP (\$xxxx) utasítással kezdődnek, ahol \$xxxx egy-egy RAM címet jelent /általában a 3. lapon/. A címek - hacsak nem irtuk át őket - közvetlenül az indirekt ugrás alá mutatnak vissza, s így folytatják a program végrehajtását. Ugyanakkor a programozónak lehetősége nyílik ezeknek a legfontosabb eljárásoknak az átirására. A legegyszerűbb lehetőség erre a CHRGET rutin átirása, amelyik a 0. lapon helyezkedik el. /A rendszer restartja esetén az első lapok értékeit az inicializáló program beállítja/. Ezekről a lehetőségekről bővebben a 10. fejezetben szólunk.

Végül pár szót szólunk a C-64 számítógép megszakító rendszeréről, amelyik közvetlenül támogatja a BASIC interpretert.

A számítógép bekapcsolása után, amikor a tápfeszültség eléri a 4.75 V-ot a számítógép automatikusan generál egy restart jelet. Ennek hatására a programszámláló az \$FFFC,\$FFFD byte-okban tárolt értékekkel töltődik fel, előtte azonban a maszkolható interapt jelzőbitje magas lesz /letiltja/. A JMP (\$FFFC) végrehajtása az I/O regiszterek feltöltésével, majd a legfontosabb rendszerváltozók értékének beállításával kezdődik. Ezt követően kerül sor a memória tesztelésére, illetve a

BASIC munkaváltozók beállítására. Legvégül a gép bejelentkezik, kiírja a szabad memóriaterület nagyságát és a READY üzenet kiírásával belép a szövegszerkesztőbe. Ez az inicializálási eljárás állítja be a BASIC interpreter megszakítási rendszerét is.

A C-64 a nem maszkolható megszakításokat nem kezeli /egy NMI jel egy JSR (\$FFFA) végrehajtását eredményezi . A maszkolható megszakítások ellenőrzésére, illetve kezelésére szolgáló rutin belépési pontja \$FF48. /Egy JSR (\$FFFE) végrehajtása után/. Ez először ellenőrzi, milyen egységekről érkezett a megszakítás, majd ettől függően egy JSR (\$314) , illetve JSR (\$316) utasítást hajt végre. Alapállapotban ezek a megszakítást kezelő ROM-rutinra mutatnak vissza, de természetesen tetszőlegesen átirhatók.

A hardver megszakító rutin, amely másodpercenként mintegy ötvenszer hajtódik végre a következő szolgáltatásokat végzi:

- a kurzor villogtatása;
- a stop billentyű lenyomásának ellenőrzése;
- a billentyűzet olvasása;
- a ϕ . lapon található óra aktualizálása;
- a megszakítások kezelése.

Részletesen erről a 8. fejezetben lesz szó.

BASIC interpreter fontosabb rendszerváltozói

(\$20)	= GOTO, GOSUB, SYS, RUN n utasítások sorszáma
(\$3B)	= az utoljára végrehajtott BASIC utasítás sorszáma
(\$BD)	= az éppen végrehajtás alatt levő utasítás sorszáma
(\$7A)	= az aktuális BASIC byte címe
\$200-\$258	= input puffer

BASIC interpreter fontosabb belépési pontok

(\$300)	= BASIC hibaüzenet kiírása
(\$302)	= BASIC meleg start
(\$304)	= input puffer kódolása
(\$306)	= ellenőrző ciklus belépési pont
(\$308)	= BASIC token kiértékelés
(\$314)	= hardver interapt vektor
(\$316)	= BRK interapt vektor
(\$318)	= nem maszkolható interapt vektor
\$A480	= BASIC meleg start
\$A49C	= programsor elhelyezése a memóriában
\$A533	= sorok mutatóinak ujraszámozása
\$A560	= input sor átírása a pufferbe
\$A57C	= input puffer kódolása
\$A7AE } \$A708 }	= ellenőrző ciklus belépési pontok
\$A7E1	= BASIC token kiértékelés
\$A871	= RUN belépési pont
(\$FFFA)	= nem maszkolható megszakítás
(\$FFFC)	= restart
(\$FFFE)	= maszkolható megszakítás

4. Fejezet

BASIC UTASÍTÁSOK

Ez a fejezet tartalmazza a BASIC utasítások leírását, működésük részletes ismertetését. Az egyes utasításokról szóló részek a következőképpen épülnek fel.

Rövidítés: A legtöbb BASIC alapszót nem kell végig kiírunk, elég az első 2-3 karakter beírása, amelyek közül az utolsót shiftelve /emelve/ írjuk be. A fenti címszó alatt a rövidítés 'kis betűk/nagy betűk' karakterkészlet használata esetén látható alakját adjuk meg.

Token: A BASIC interpreter az alapszavakat tokenjükkal tárolja. Az érvényes tokenek 128 és 203 közé esnek. A fenntartott változóknak /ST, TI, TI\$/ nincs külön tokenje.

Belépési pont: A megnevezés kicsit félrevezető. Az interpreter a 'belépési pont'-ban megjelölt érték alsó és felső byte-ját a verembe tölti, majd végrehajt egy RTS utasítást. Ennek hatására a processzor a verem tetején levő két byte-ot betölti az utasításszámlálóba, majd az így kapott értéket megnöveli 1-gyel. Ettől a cimtől folytatódik a program futása. Például a VAL belépési pontja: \$B7AD (47021). A helyes vezérlésátadást a következőképpen érhetjük el:

BASIC-ből: SYS 47021+1

Gépi kódból: LDA #\$B7, PHA, LDA #\$AD, PHA, RTS vagy
 JSR \$B7AE (=\$B7AD+1) .

A könyv többi részében a helyes vezérlésátadás SYS (BELEPESI-PONT).

Funkció: A bevezető részben röviden összefoglaljuk az utasítás hatását. A BASIC ismerők számára ez általában elegendő az utasítás használatához. Kezdők feltétlenül tanulmányozzák át a példákat, illetve a végrehajtást leíró részt.

Szintaxis: Az egyes szintaktikus egységek megnevezéseit csu-
csos zárójelek közé tesszük; például <aritmetikai kifejezés>
A szögletes zárójelek közti részek nem kötelezőek, opcionáli-
sak, például

LIST [- <sorszám>]

A kapcsos zárójelekben, egymás alatt szereplő részek kötelező
választási lehetőségeket sorolnak fel /lásd például az ON uta-
sításnál./ A szintaxis leírásában szereplő többi jel (", :
stb.) az utasítás része. A szintaxis leírása a mondat végét
jelentő ponttal (.) fejeződik be. A pont természetesen nem ré-
sze az utasításnak.

Példák: A példákat úgy igyekeztünk összeválogatni, hogy az uta-
sítások legtipikusabb használati lehetőségeit bemutassák. A
példákat követő megjegyzések segítenek az utasítás pontos hata-
sának megértésében.

Végrehajtás: Itt írjuk le, hogy az interpretér hogyan is hajtja
végre az utasítást. Ez elsősorban azok számára segítség, akik
az utasítások végrehajtásával gépi-kódu szinten szeretnének meg-
ismerkedni.

Hibalehetőségek: Itt soroljuk fel a leggyakoribb hibákat.

4.1 BASIC alapszavak ABC sorrendben

ABS

Rövidítés: aB Token: \$BA (182)

Belépési pont: \$BC59 (48217)

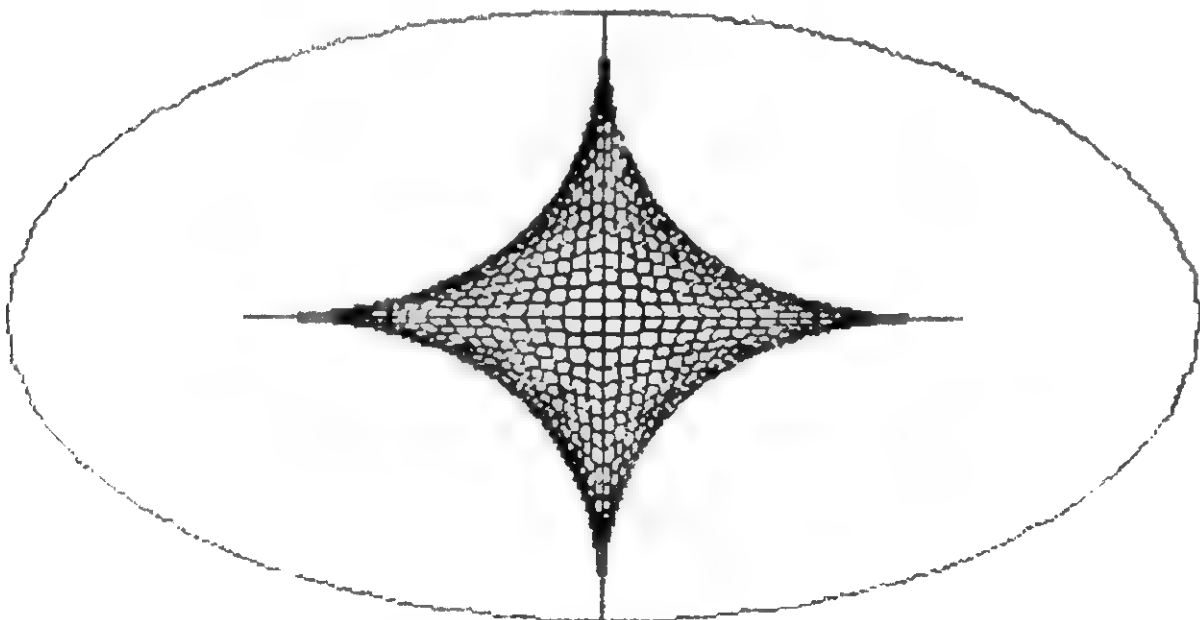
Mód: Mind parancs, mind program módban használható.

Az ABS jelet zárójelben követő aritmetikai kifejezés abszolút értékét számolja ki. Ennek következtében ABS mindig egy nem negatív értékkel tér vissza. Elsősorban numerikus feladatok megoldása során használjuk.

Szintaxis: ABS (< aritmetikai kifejezés >)

```
10 IF ABS(X-X1)<EPS THEN PRINT "** VEGE **"
20 IF ABS(QY)>1E7 THEN PRINT "** QY TUL NAGY **"
30 IF ABS(X%)<3 THEN GOTO...
```

READY.



Az első példa egyenletek közelítő megoldásában igen gyakori. Ellenőrzi, hogy a gyök x és x_1 közelítései elég közel vannak-e egymáshoz. Ha igen, a program megáll, különben tovább folytatja a számítást.

A második példa az ABS direkt módban való használatát szemlélteti. A program elindítása előtt ellenőrizzük QY értékét, ha az meghaladja az 1E7 értéket a gép egy **** QY TUL NAGY **** üzenet kiírásával figyelmeztet. Az üzenet természetesen magába a programba is beépíthető.

A harmadik példa egy NIM játékprogramból való. A gép véletlenszerűen generál gyufaszálból álló kupacokat. Ha túl kevés $/ < 3 /$ gyufa van egy kupacban újból generál egy számot.

Végrehajtás: A zárójelben szereplő kifejezés értékét számolja ki először, majd betölti az **# 1** lebegőpontos akkumulátorba. ABS csak az előjelet tartalmazó byte-ot **/\$66,102/** változtatja meg, eltolja jobbra, így annak 7. bitje \emptyset lesz. A művelet végrehajtása a szám pontosságát nem befolyásolja. Kerekítési hibák csak az argumentum kiszámításánál fordulhatnak elő. Például **ABS(-12456789101)** és **ABS(123456789101)** értéke megegyezik de nem egyenlő 123456789101-gyel.

Hibalehetőségek: Ha a zárójelben nem, vagy rossz aritmetikai kifejezés van különféle hibajelzések generálódhatnak, például szintaktikus hiba, nem megfelelő típus jelzése, stb. Ha az aritmetikai kifejezés kiértékelése tulcsordulást okoz az egy **?OVERFLOW ERROR** végrehajtását eredményezi.

C-64

4.5

AND

kétváltozós logikai művelet

Rövidítés: aN Token: \$AF (175)Belépési pont: \$AFE9 (45032)Mód: Mind parancs, mind program módban használható.

Két kifejezés logikai 'és'-ét határozza meg, előbb azonban ezeket 2-byte-os egész számmá konvertálja. Az 'és' műveletet ezután bitenként, egymástól függetlenül hajtja végre. Az eredmény maga is egy 2-byte-os egész szám. Ha mindkét kifejezés logikai volt, akkor eredményül 0 /'hamis'/ vagy - 1 /'igaz', \$FFFF/ adódik.

Szintaxis: <aritmetikai vagy logikai kifejezés > AND
<aritmetikai vagy logikai kifejezés > .

Példák:

```
10 PRINT 300 AND 75
20 OK=X>Y AND Y+Z>20 AND X>15
30 IF A<12 AND B*CC>16 THEN GOTO ...
```

READY.

Az első példában azt mutatjuk be mi történik, ha aritmetikai kifejezéseket kapcsolunk össze AND-del. A két szám 16-bites megfelelője: -241

/= %1111111100001111/ és 15359 /=%00111011111111111/. Az 'és' műveletet bitenként elvégezve %0011101100001111, azaz 72 adódik. Az első példa végül is 72-öt nyomtat ki.

A másik két példa azt mutatja, hogyan használhatjuk az AND művelet összetett állítások képzésére és tesztelésére.

A második példában az OK egész változó értéke - 1 lesz abban az esetben, ha $X > Y$, $Y + Z > 20$, illetve $X > 15$ egyaránt teljes. Különben OK értéke \emptyset lesz.

Az utolsó példa az AND használatát egy összetett feltételes utasításban szemlélteti. Az üzenet csak abban az esetben íródik ki, ha mind a két feltétel teljesül.

Végrehajtás: A két kifejezés értéke az #1 illetve a #2 lebegőpontos akkumulátorba töltődik. Egy jelzőbyte /TEST/ értéke $\$00$ -val töltődik fel. Az akkumulátorok tartalmát 2 byte-os egész számmá konvertálja, majd a következő összetett függvényt számítja ki:

TEST EOR ((TEST EOR A) AND (TEST EOR B)) .

Az OR-t végrehajtó rutin ugyanezt az eljárást használja, de akkor a jelzőbyte értéke ~~#~~\$FF.

Amikor TEST értéke $\$00$, akkor TEST EOR X = X

és így

A AND B = ... (... A AND ... B)

A 2-byte-os egészek alsó és felső byte-jaira a fenti műveletet egymástól függetlenül hajtja végre.

Hibalehetőségek

/i/ Ha az operandusok kiértékelése közben hiba történik, a hiba típusának megfelelő jelzést kapunk. Ha az akkumulátorok tartalmának konvertálása túlcsordulást okoz, akkor ?CVERFLOW ERROR üzenetet kapunk.

/ii/ Gyakori hiba forrása a logikai műveletek sorrendjének be nem tartása. A C64 BASIC műveleti hierarchiája megegyezik a FCRTAN és ALGOL programnyelvekével.
Először a NOT, azután az AND, legvégül az OR hajtódik végre.

/iii/ Először az aritmetikai műveleteket hajtja végre a gép, majd a logikaiakat. Ha például egy X számból le akarunk vonni 1-et, ha $2 \nmid Y$ és $Z \leq 6$ ezt gyakran a következőképpen írják fel:

$$XUJ = X + \left(\text{INT} \left(Y/2 \right) * 2 = Y \right) \text{ AND } Z \leq 6$$

A végrehajtási sorrend miatt először az

$X + \left(\text{INT} \left(Y/2 \right) * 2 = Y \right)$ érték számítható ki és azután az AND művelet. A helyes megoldás:

$$XUJ = X + \left(\left(\text{INT} \left(Y/2 \right) * 2 = Y \right) \text{ AND } \left(Z \leq 6 \right) \right)$$

/iv/ A fenti programrészt több részletben is kiszámíthatjuk. Például a zárójeles részt a következő utasítással

$$XD = \text{INT} \left(Y/2 \right) * 2 = Y \quad \text{AND} \quad Z \leq 6$$

A C64 BASIC a fenti sort a következőképpen értelmezi:

C-64

4.8

$$XD = INT \ (Y/2) * 2 = YA \ < \ = 6$$

/v/ A matematikában gyakran használjuk például három szám egyenlőségének jelzésére az al és bl = cl jelölést. Hasonló rövidítés a BASIC-ben nem megengedett. Például a memóriában két egymás utáni \emptyset byte-ot kereső program részeként

```
IF PEEK ( c ) AND PEEK ( c + 1 ) = 0 THEN..
```

nem éri el a kívánt hatást. Helyette

```
IF PEEK ( c + 1 ) = 0 AND PEEK ( c ) = 0 THEN...
```

vagy

```
IF PEEK ( c + 1 ) + PEEK ( c ) = 0 THEN ...
```

szükséges. Az utóbbi csak azért működik helyesen, mert a PEEK-elt értékek nem negatívok.

ASC

aritmetikai függvény

Rövidítés: aS Token: \$C6 (198)

Belépési pont: \$ B786 (46988)

Mód: Mind parancs, mind program módban használható.

Az ASC jelet a zárójelben követő sztring kifejezés első karakterének C64 ASCII kódját adja. ASC használata gyakori azokban az esetekben, amikor egy ASCII karakter kódját, ami egy szám, könnyebb kezelni, mint magát a karaktert.

Szintaxis: ASC (< sztring kifejezés >)

C-64

4.9

Példák:

```

10 GET JEL$: IF JEL$="" GOTO 10
20 JEL=ASC(JEL$): IF JEL=13 THEN RETURN
30 IF JEL=20 THEN ....
99 :
100 X=ASC("ZEBRA")
199 :
200 JEGY=ASC(KOD$)-48+(ASC(KOD$)>64)*7

```

READY.

Az első példában szereplő programrészlet azt mutatja, hogy az ASC függvény segítségével a billentyűzetről kapott információit /JEL\$/ hogyan elemezhetjük. Ezzel az eljárással a teljes billentyűzetet ellenőrizhetjük, kivéve a STOP gombot.

A második példa eredményül 90-et ad. Z ASCII kódja ugyanis 90.

Harmadik példánk egy hexadecimális - decimális konvertáló program része. Ha a KOD\$ első karaktere számjegy, akkor JEGY értéke ez a számjegy lesz. Ha például KOD\$ = B89A , akkor JEGY = 11 a B "számjegy" értéke.

Az első példában az ASC függvény helyettesíthető a CHR\$ függvénnyel. Például a JEL = 13 helyett rögtön a JEL\$=CHR\$ 13-t kérdezhetnénk. Ekkor persze JEL kiszámítása teljesen felesleges.

Végrehajtás Először az ASC-t követő sztring kifejezést értékeli ki az interpreter, majd a verembe betölt 3 byte-ot /az eredményül kapott sztring hossza, 2-byte-os pointer a

C-64

4.10

sztring első elemzése/. Az ASC végrehajtása ezután kezdődik. ViSSzatölti a veremből a paramétereket, és ellenőrzi a sztring hosszát. Ha ez 0, akkor ?ILLEGAL QUANTITY ERROR jelzést ad. Ha nem, a sztring első elemét betölti az akkumulátorba. Ennek értéke éppen a keresett ASCII érték. Egy konverziós rutin ezt lebegőpontos számmá alakítja és betölti az #1 akkumulátorba.

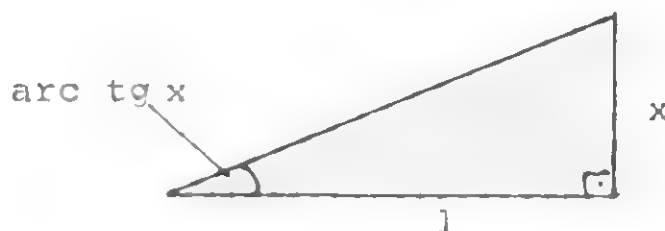
Hibalehetőségek A kifejezés kiértékelése közben számos hiba fordulhat elő, köztük, hogy az eredménystring hossza meghaladja a 255-öt. Ha a kifejezés nem sztring típusu, akkor "nem megfelelő típus" üzenetet kapunk. Gyakori hiba, hogy ASC-t hajtunk végre egy nulla hosszúságú sztringre. Ezt elkerülhetjük ha $ASC(J\$ + CHR\$(0))$ -t használunk $ASC(J\$)$ helyett.

ATN

aritmetikai függvény

Rövidítés: aTToken: \$AF (175)Belépési pont: \$E3OE (58126)Mód: Mind parancs, mind program-módban használható.

Az argument arkusz tangensének főértékét számítja ki radiánban. Az alábbi ábra illusztrálja x és $\arctan x$ kapcsolatát:



C-64

4.11

Szintaxis: ATN (<aritmetikai kifejezés>) .

Példák:

```
1100 ALFA=-ATN(YV/ZV):BETA=-ATN(XV/YV)
2100 LET R=ATN((Y2-Y1)/(X2-X1))
```

READY.

Mindkét példa geometriai jellegű. Az első programsor az /XV,YV,ZV/ pontnak a síkon való ábrázolásakor a tengelyekkel bezárt szögét számítja ki. A második programsor egy derékszögű koordinátarendszerből, polárkoordináta rendszerbe átszámító alprogram része. A fenti példákban ALPHA, BETA és R értéke a $-\pi/2$, $\pi/2$ intervallumba esik. Ha ezt az értéket megszorozzuk $180/\pi$ -vel megkapjuk az eredményt szögekben is.

Végrehajtás: A kifejezés értékének kiszámítása után annak előjelét külön tárolják, valójában $\arctg|x|$ kerül kiszámításra. A konkrét számítást egy 12-tagú hatványsor végzi. Ez azonban csak a 0-1 intervallumban dolgozik helyesen. Ha $|x|$ nagyobb mint egy, $\arctg \frac{1}{|x|}$ értéket számítja ki az interpreter. A helyes eredményt úgy kapjuk meg, hogy ezt az értéket kivonjuk $\pi/2$ -ből. /Tudva, hogy $\alpha + \beta = 90^\circ$ esetén $\tg \alpha \cdot \tg \beta = 1$ / Mivel az interpreter a COS, SIN, LOG függvényeket is hatványsor segítségével számítja ki, külön rutin végzi egy tetszőleges hatványsor kiszámítását.

Hibalehetőségek: Az aritmetikai kifejezés kiértékelése közben számos hibalehetőség adódik. Ha a kifejezés kiértékelhető és nem haladja meg a gépben ábrázolható valós számok felső határát, akkor az ATN rutin maga már hibajelzés nélkül végrehajtódik.

C-64

4.12

CHR\$

sztring függvény

Rövidítés: CH /a \$ már nem kell!/
Token \$C7 (199)Belépési pont: \$B6EC (46828)Mód: Mind parancs, mind program-módban használható.

Egy tetszőleges, a [0,255] intervallumba eső számból előállít egy 1 hosszúságú sztringet, amelyik egyetlen karakterének ASCII kódja éppen a szóbanforgó szám. A C-64 BASIC-ben ez az egyetlen kényelmes eljárás bizonyos speciális karakterek /például [RETURN] és "/ kezelésének /a példában CHR\$ (13) és CHR\$ (34) /.

Szintaxis: CHR\$ (< aritmetikai kifejezés >)

Az aritmetikai kifejezés nem kell, hogy egész legyen. Ilyenkor a kifejezés értékének veszi az egész részét, s ezután ellenőrzi, hogy beleesik-e a jelzett intervallumba. Például CHR\$ (-.01) , CHR\$ (400) és CHR\$ (X\$) használata hibát eredményez.

Példák:

```
10 A$=CHR$(34)+CHR$(18)+"NEV"+CHR$(146)+CHR$(34)
20 XE$=CHR$(160)+X$
30 PRINT#4 CHR$(7)
40 PRINT#4 CHR$(27)
```

READY.

Példáink azt mutatják be, hogyan lehet a CHR\$-t speciális szerkesztő karakterek beillesztésére felhasználni. Ezeket a karaktereket különben igen nehéz előállítani. Az első példa idézőjelek közé helyez egy sztringet, előtte azonban eléje és utána helyezi a [RVSON] illetve a [RVSOFF] karaktereket. ?A\$ hatására idézőjelek közt jelenik meg a NAME negatív /inverz/ képe. A 2. példa az X\$ sztring elejére egy siftelt szóköz karaktert illeszt, az jobban olvasható mint az XE\$ = " " + X\$ változat. A harmadik példa elküld a nyomtatónak egy 'bell' jelet. Bizonyos típusu nyomtatók ilyenkor valóban adnak valamilyen hangjelzést. A negyedik példa ugyancsak a nyomtatóhoz kapcsolódik; a vezérlő karakter hatására dupla-széles karaktereket nyomtat a printer. Az ötödik példa a RAM-területről másol a képernyőre. Nem a legszerencsésebb megoldás, hiszen a vezérlő karakterek igencsak furcsa képernyőt eredményezhetnek /pl. egy [CLEAR] ; CHR\$ (147) /. Az utolsó példa a C\$ sztringben gyűjti össze a memóriában egymás után elhelyezett hat karaktert.

A CHR\$ sztringfüggvény az ASC inverze. Speciális felhasználási területe a különböző karakterkészletek közti konverzió. A PRINT utasítást használó programok nem írhatóak át közvetlenül PRINT * 4-re Előbb a PEEK-elrt értékek részletes vizsgálata szükséges, hogy a nyomtatási kép megegyezzen a képernyővel.

CHR\$ (0) egy null karaktert jelent, de a hossza 1. A nyomtatási képben ennek azonban nincs nyoma. Legyen például Y\$ = "123" + CHR\$ (0) + "321". A ? Y\$ utasítás hatására az 123321 sorozat íródik ki, de LEN (Y\$) = 7, VAL (Y\$)=123.

C-64

4.14

Végrehajtás: Először az argumentum értékét számítja ki, azután ellenőrzi, hogy egész része 0-255 közé esik-e. Ha igen, akkor egy 1 hosszúságú sztringet helyez el a memóriában. Ha a sztring egy értékadásban szerepel, például `X$ = CHR$ (34)`, akkor ez a sztring megmarad a további számítások során is, ha nem, például a `? CHR$ (34)` utasításban, akkor a sztring-terület mutatóit nem állítja át, s a következő sztring felülírja a karaktert.

Hibalehetőségek: Az argumentum kiértékelése és nem megfelelő értéke számos hibajelzést eredményezhet. A speciális karakterek kódjainak rossz használata esetén a kívánt hatás elmarad.

CLOSE

Rövidítés: `clo`

Token: `$AO (160)`

Belépési pont: `$FFC3 (65475)`

Mód: Mind parancs, mind program módban használható.

Befejezi egy adott file feldolgozását, törli a file-t a három file-leíró táblából. A billentyűzetre és a képernyőre megnyitott file-okkal a törlésen kívül semmi egyebet sem csinál. A többi file esetében a tevékenység attól függ, milyen utasítással nyitottuk meg. Ha egy kazettás file-t olvasásra nyitottunk meg, akkor semmilyen további tevékenységet nem végez az interpreter, ha viszont írásra, akkor az OPEN utasításban szereplő 'megnyitási mód' paramétertől függően még egy 'END OF TAPE' jelet is kiír /ha a paraméter 2 volt/. Ha egy lemezen levő file-t írásra is megnyitottunk, akkor egy 'END OF FILE' jelet ír be a lemezes file utolsó szektorába.

C-64

4.15

Szintaxis: CLOSE <aritmetikai kifejezés>. Az utasítás fenti formájában a logikai file-számot további paraméterek követhetik, ezeket azonban az utasítás nem használja semmire.

Példák:

```
10 OPEN 4,4 : PRINT#4,"HOGY VAGY?":CLOSE4
20 OPEN 1,1,1,"FILE":PRINT#1,"HOGY VAGY?":CLOSE1
100 CLOSE 1,2,3,"$": REM UGYANAZ, MINT CLOSE 1
1000 PRINT#8,CHR$(13)::CLOSE8: REM LEMEZEK FILE LEZARASA
1100 PRINT#4,::CLOSE 4: REM LEZARJA A NYOMTATOT CMD UTAN
```

A CLOSE utasítás a CMD és a PRINT# utasításokkal együtt használva igen nehézkes. A CMD esetében az output eszköz még mindig 'hallgató', és ahhoz, hogy ezt az állapotot megszüntessük, egy további PRINT# utasítás szükséges. A PRINT utasítás CBM lemezmeghajtók esetében legalább is még egy soremelés /CHR\$(10)/ karaktert ír a kocs-vissza-soremelés karakter után, amelyik az adott rekord végét jelzi.

Végrehajtás: A paramétereket ugyanaz a rutin értékeli ki, mint az OPEN utasítás esetében. /Ezért lehetnek a CLOSE-nek további paraméterei!/. Az interpreter ellenőrzi, hogy a logikai file-szám a 0-255 intervallumba esik-e, majd megvizsgálja a file milyen I/O egységhez tartozik. Az egységszám határozza azután meg a további teendőket.

Hibalehetőségek: Elsősorban írásra megnyitott lemezes file-ok esetén lényeges, hogy ezeket a file-okat lezárjuk. Ha ezt nem tesszük meg, az utolsó rekord track/sector mutatója nem megfelelő helyre mutat, s előbb vagy utóbb két file

egymásra másolódik. Programhiba következtében gyakran maradnak nyitva file-ok. Ilyenkor célszerű a file-okat paranccsal lezárni. Előfordulhat, hogy 'megnyitott—fileok-száma' mutató /\$ 98, 152/ nulla lesz /pl. szerkesztés után/. A file-t még ekkor is megkísérelhetjük lezárni a

```
POKE 152,10 : CLOSE X%
```

parancs kiadásával. Természetesen, ha a táblák is törlődtek, akkor semmit sem tehetünk.

CLR

Rövidítés: cL

Token: \$9C (150)

Belépési pont: \$A65D (42589)

Mód: Mind parancs, mind program-módban használható. Valamennyi egyszeri és tömb változót töröl a memóriából, magát a BASIC programot azonban változatlanul hagyja.

Szintaxis: CLR

Példák:

```
10 CLR
20 CLR:PRINT"A VALTOZOKAT TOROLTUK"
30 POKE 55,0:POKE 56,48:CLR:REM A MUNKATERULET VEGE $3000
```

READY.

A példák önmagukért beszélnek. Bármilyen is volt a változók értéke eddig, a program vagy parancs folytatása után kezdőértékükkel használhatjuk őket. A 3. példában a BASIC munkaterület végét jelző byte-okat átállítottuk. Ilyen esetben a CLR mindenképp szükséges, hiszen az átállítás után a legtöbb BASIC mutató már használhatatlan.

Végrehajtás: A 'sztring változók kezdete' mutatót az interpreter a 'BASIC terület végére' állítja, majd ezt követően a 'tömb változók kezdete', tömb változók vége' mutatókat közvetlenül a tárolt program utáni első szabad helyre állítja át. A BASIC változók tehát nem törlődnek abban az értelemben, hogy mondjuk, minden byte-juk helyére \emptyset kerül, pusztán az interpreter számára válnak felismerhetetlenné.

Ezután kerül sor egy RESTORE végrehajtására, amelyik az 'aktuális DATA sor' mutatót a program elejére állítja. Egy KERNAL rutin hívásával /\$FFE7 a belépési pont/ valamennyi I/O tevékenységet abortálja, elsődleges input illetve output file-nak a billentyűzetet, illetve a képernyőt állítja be. A CLR utasítás végrehajtásának utolsó lépéseként a verem tetejét jelző mutatót is visszaállítja kezdőértékére. Ennek következtében a FOR... NEXT illetve a GOSUB...RETURN hivatkozások elvesznek. A verem nem ürül ki teljesen, mert a visszatéréshez szükséges cím a verem tetején megmarad. Amikor a CLR-t végrehajtó rutin végrehajtja a RTS utasítást, a vezérlés a megfelelő helyre adódik vissza.

Hibalehetőségek: Az utasítás, helyes szintaxis esetén, mindig hiba nélkül végrehajtódik. Problémát csak az okozhat, hogy egyes változók értékei - amelyekre később szükség lenne - elvesznek. A CLR semlegesítése igen nehéz, ezért óvatosan bánjunk vele !

CMD

Rövidítés: CM

Token: \$9D (157)

Belépési pont: \$AA85 (43653)

Mód: Mind parancs, mind program módban használható.

C-64

4.18

A CMD utasítás két különböző funkciót egyesít:

- /i/ Segítségével módosíthatjuk az elsődleges output file-t; a CMD utasítást követő PRINT utasítások a CMD-ben megadott file-ba íródnak, bármi legyen is az.
- /ii/ A CMD utasításban megadott sztringet kiírja az új output file-ba.

Szintaxis: CMD <aritmetikai kifejezés> [,<nyomtatási kép>]

Példák: Tegyük fel, hogy egy OPEN 5,4 utasítással megnyitottuk a nyomtatót. A CMD utasításokban a nyomtatóra természetesen az 5-ös logikai file-számmal hivatkozhatunk.

[i]

```
10 CMD 5          A TOVABBI OUTPUT A PRINTERRE MEGY.
20 CMD 5,;        A TOVABBI OUTPUT A PRINTERRE MEGY CRLF NELKUL
30 CMD 5,"HELLO"
40 PRNT=5:CMD PRNT : REM SZINTAKTIKUSAN HELYES
```

READY.

[ii]

```
OPEN 4,4 : CMD 4: INPUT"NEV";K$
[iii]
10 OPEN 4,4:CMD,;
20 PRINT "EREDMENYEK"
30 GET K$: IF K$="" THEN GOTO 30
40 PRINT#4,;:CLOSE4:END
```

Az [i] alatt felsorolt példák önmagukért beszélnek. Ha azonban a CMD 5, "HAHO!" utasítást összehasonlítjuk a PRINT 5, "HAHO!" utasítással. rögtön világossá válik, mennyire hasonlóak. Zavaró azonban, hogy míg a CMD5,; pa-

ranos hatására a nyomtató 'hallgató' állapotba kerül, addig a PRINT # 5,; hatása ennek pont az ellenkezője; az output eszköz megszűnik 'hallgató' lenni.

A [ii] példa a CMD egy igen szerencsétlen hatását mutatja be; az input prompt a CMD-ben specifikált file-ba íródik, nem a képernyőre. Hasonló a célja a [iii] példának is. A GET hatására az elsődleges output file újból a képernyő lesz. A 40. sor mutatja, hogyan kell a CMD-ben használt file-t helyesen lezárni.

Megjegyezzük, hogy rendszerhiba esetén a CMD hatása megszűnik, és az elsődleges output egység újból a képernyő /VDU/ lesz. A fenti negatív hatások miatt a CMD utasítást főleg listázásra célszerű használni, programban kellemesebb a PRINT# utasítás használata.

Végrehajtás: A CMD utáni paraméter kiszámítása és ellenőrzése /1-255 közt kell, hogy legyen/ utána a buszra olyan utasítás kerül, amelyik 'hallgatóvá' teszi a megfelelő egységet. Ehhez az OPEN-táblából keresi ki az interpreter a megfelelő adatokat. Ha ez valamiért nem sikerül, akkor ?FILE NOT OPEN vagy ?DEVICE NOT PRESENT hibaüzenetet kapunk. Ezután kerül sor a második paraméter ellenőrzésére. Ha van, a PRINT utasítás megfelelő része kerül végrehajtásra.

Hibalehetőségek: A példák során már utaltunk a CMD utasítás néhány furcsaságára. Célszerű csak listázásra használni.

CONTRövidítés: cOToken: \$9A (154)Belépési pont: \$A856 (430%)Mód: Csak parancs-módban használható.

A program továbbindítására szolgál, miután a program futása egy STOP/END végrehajtása után, vagy a STOP billentyű benyomása miatt megszakadt. A STOP felhasználásával ellenőrző pontok iktathatók be a programba, majd a megállás után a CONT-tal folytatható.

Szintaxis: CONT. Egyéb paraméterek nem szerepelhetnek, a CONT abban a sorban, vagy az utolsó utasítás , vagy kettőspont követi.

A BASIC program futása közben, a program végrehajtását ellenőrző ciklus állít be bizonyos mutatókat, amelyek segítségével a CONT végrehajtható. Ezek a következők:

/ \$39/ = a feldolgozás alatt álló BASIC sorszám.

/ \$3B/ = az utoljára végrehajtott BASIC utasítás sorszáma

/ \$3D/ = a BASIC szöveg következő feldolgozandó byte-jára mutat /CONT itt folytatja/

A CONT utasítás támogatása természetesen lassítja a program futását. A program megállása után a változók értékei kiírhatók és módosíthatók parancs-módban. A CONT még ezután is végrehajtható. Néhány esetben /pl. CLR, NEW, ?SYNTAX ERROR után/ a program futása nem folytatható. Ezek az utasítások / \$3E/-be 0-t helyeznek. Ha a / \$3D/ mutató értékét jól állít-

C-64

4.21

juk be, a BASIC program bármelyik helyéről elindíthatjuk a programot a CONT paranccsal.

Végrehajtás: Először a CONT utasítás szintaxisát ellenőrzi az interpreter, majd ellenőrzi, hogy a /\$3D/ magas byte-ja nulla-e, vagy sem. Ha nullát talál, ?CAN'T CONTINUE hibajelzéssel megáll. Ha egy érvényes mutatót talál, átállítja a CHRGET és az 'aktuális BASIC sorszám' mutatót és visszatér az ellenőrző ciklus elejére.

Hibalehetőség: NEW, CLR, ?CAN'T CONTINUE, szerkesztés után a program futása nem folytatható. Ilyenkor GOTO-val célszerű kísérletezni.

COS

aritmetikai függvény

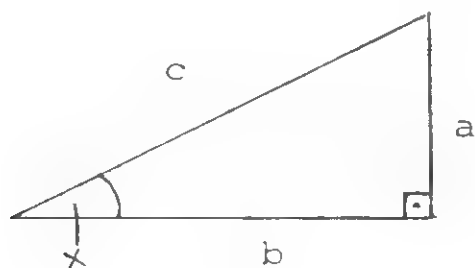
Rövidítés: nincs

Token: \$BE (190)

Belépési pont: \$E264 (57956)

Mód: Mind parancs, mind program-módban használható.

A radiánban megadott szög koszinuszát számítja ki. Az alábbi ábra illusztrálja x és $\cos x$ viszonyát:



$$\cos /x/ = \frac{b}{c}$$

C-64

4.22

Szintaxis: $\text{COS}(\langle \text{aritmetikai kifejezés} \rangle)$. A kifejezésnek szintaktikusan helyesnek kell lennie.

Példák:

```
10 PRINT COS(1) :REM 1 RAD = .54 FOK
20 PRINT COS(90*PI/180): REM ATSZAMITAS FOKBA
3010 Y=EXP(-L*T)*(A*SIN(T)-B*COS(T))
3020 X=ALFA+SIN(ALFA):Y=1-COS(ALFA)

READY.
```

Az első két példa önmagáért beszél. A [3] , [4] példákban speciális trigonometrikus függvények képeit számítjuk ki. A [4] például egy cikloid X, Y koordinátáit számítja ki.

A szög nagysága a végeredményt nem befolyásolja, a kiszámítás során ugyanis az argumentumot az interpreter elosztja $\pi \times 2$ -vel és a maradékkal helyettesíti.

Végrehajtás: az aritmetikai kifejezés értékét a rutin az $\pi/2$ lebegőpontos akkumulátorba tölti, majd hozzáad $\pi/2$ -t. Ezután végrehajtja a SIN alprogramot, a $\text{COS}(X) = \text{SIN}(X + \pi/2)$ azonosság alapján.

Hibalehetőségek: Az aritmetikai kifejezés kiértékelése közben számos hibalehetőség adódik. Ha az értékelés befejeződött, maga a COS rutin már hibajelzés nélkül végrehajtódik.

C-64

4.23

DATARövidítés: dAToken: \$83 (131)Belépési pont: \$A8F7 (43255)Mód: Csak program-módban használható.

A fenti utasítás lehetővé teszi, hogy a programban számokat, sztringeket tároljunk, s ezeket a READ utasítás segítségével beolvashassuk. A READ abban a sorrendben olvassa be ezeket az adatokat, ahogy azok a DATA utasításban követik egymást.

Szintaxis: DATA < konstanslista > . A konstanslista elemei egymástól vesszővel /,/ elválasztott konstansok. Mind szöveg /sztring/, mind számkonstansok szerepelhetnek a listában. A szövegkonstansokat nem kell idézőjelek közé tenni. /Kivéve ha tartalmaz vesszőt (,).

Példák:

```
102 DATA "KOVACS PETER, 1957","NAGY ISTVAN, 1964"
104 DATA "ERDOS ILONA, 1965"
110 :
1000 DATA GEPI KOD,120,169,46,133,96
1999 :
2000 DATA 1,1,2,2,3,0 : MUVELETEK RANGJA
2010 DATA 0,1,2,3,4,5,6,7,8,9:ERVENYES SZAMJEGYEK
2020 DATA +,-,*,/,↑,π :MUVELETI JELEK
```

READY.

Az első példa sztringek DATA-ban való tárolását mutatja. A

```
FOR I = 1 TO 3 : READ X$ : ?X$ : NEXT
```

utasítással mind a három szövegkonstans kiiratható. A második példa illusztrálja, hogyan lehet bizonyos adatokat meg-

találni a DATA-k közt. Addig hajtjuk végre a READ X\$ utasítást, amíg X\$ ="GEPI KOD" nem teljesül. A harmadik példa mutatja, hogy a DATA-kban célszerű strukturáltan elhelyezni az adatokat, hogy könnyebben cserélhetők legyenek.

Végrehajtás: A DATA utasítás végrehajtása az utasítás kihagyását jelenti azzal a különbséggel, hogy nem az egész programsor marad ki /mint a REM esetén/, hanem csak az utasítás. A rutin megkeresi a következő kettőspontot vagy null byte-ot és onnan folytatja a program végrehajtását.

Hibalehetőségek: A READ utasítás néhány rutinja megegyezik az INPUT utasítás hasonló rutinjaival. Emiatt a nem siftelt, sztring elején álló szóközök és bizonyos grafikus jelek elvesznek. ?SYNTAX ERROR hibaüzenet egy DATA utasítással kapcsolatban általában a hozzá tartozó READ utasítás hibáját jelenti. READ X\$ sohasem okozhat problémát /kivéve, ha nincs több adat/, READ X esetén nem mindegy, mi a következő adat. Felesleges vessző is okozhat problémát. A DATA 1,2,3 utasítás négy konstanst tartalmaz, a negyedik egy null sztring.

DEF FN

Rövidítés: dE /fn-nek nincs rövidítése/, Tokens: DEF \$ 96 (150)
FN \$A5 (165)

Belépési pont: \$B3B2 (46003)

Mód: Csak program-módban használható

A DEF FN utasítás segítségével aritmetikai függvényt definiálhatunk, amelyre azután az FN segítségével hivatkozhatunk. A függvény definíciójának van egy neve, és egy változója.

Szintaxis: DEF FN <valós változó> (<valós változó>) = <aritmetikai kifejezés> . Az első <valós változó> a függvény neve, a második a 'változója'. A függvényt definiáló aritmetikai kifejezés legfeljebb egy-somyi lehet. A függvény definiálása után az FN <valós változó> (<aritmetikai kifejezés>) újabb aritmetikai kifejezés lesz, ami a programban bárhol használható.

Példák:

```
10 DEF FN DEEK(X)=PEEK(X)+256*PEEK(X+1)
100 DEF FN MIN(X)=(A+B)/2-ABS(A-B)/2
110 DEF FN MIN(X)=-(A>B)*B-(B>=A)*A
1000 DEF FN F(X)=A*X*X+B*X+C
```

READY.

Az első példa egy 'dupla pontosságú PEEK'-et definiál. Ha két egymás utáni byte egy cím alsó illetve felső byte-ját tárolja /ebben a sorrendben/, akkor a PEEK /x/ függvény ennek a címnek az értékét számítja ki.

A második példában két megoldást láthatunk a minimum függvény kiszámítására. Ez nem igazi függvény, hiszen az X csak ál-változó /dummy variable/, csak a szintaxis kedvéért szerepel. Az A és B értéket a függvény kiszámítása előtt meg kell hívni, valahogy így

```
A = 12; B = 24; ?FN MIN (X)
```

X helyén bármi szerepelhetne FN MIN /0/, FNMIN /10/ értéke megegyezik.

A harmadik példánk egy másodfoku függvény helyettesítési értékeit számítja ki, a használat előtt A,B,C értékét definiálni kell.

A függvény definíciók további függvénydefiníciókat is tartalmazhatnak. Például

```
DEF FN EX(X) = 1 + X + X ↑ 2/2 + X ↑ 3/2/3+.. FN E(X)
DEF FN E.(X) = X ↑ 6 /1/2/3/4/5/6
megengedett.
```

A függvények, hasonlóan más változókhoz újradefiniálhatóak
 DEF FN Y/X/=Y : DEF FN Y/X/ = X helyes.

Végrehajtás: A DEF végrehajtása a mód, az FN token, a változók típusa, a zárójelek és az '=' jel ellenőrzésével kezdődik. A kifejezés szintaxisát a rutin nem ellenőrzi. Ezt követően a változók közé felkerül a következő 7 byte.

NÉV	NÉV	KIF.	MUT.	VÁLT.	MUT.	Ø
ASCII+128	ASCII	ALSÓ	FELSŐ	ALSÓ	FELSŐ	

A változó nevének első karaktere kerül az első byte-ra, azzal a különbséggel, hogy a 7. bitet a rutin magasra /1/ állítja. Így nem keverhető össze más típusu, ugyanilyen nevű változóval.

Megjegyezzük, hogy a függvény kiértékelése közben a függvény definíciójában szereplő változó értéke nem változik. Az első példa PEEK függvényét ha használjuk, akkor X értéke nem változik. A függvény kiértékelésekor X-et ideiglenesen tárolja az interpreter, majd visszaállítja az eredeti értékét.

Hibalehetőségek: Ha a DEF után az egyenlőségig valamilyen hibát követünk el, akkor ?SYNTAX ERROR hibajelzést kapunk. Ha az aritmetikai kifejezés kiértékelése közben történik hiba, akkor a hibajelzést a megfelelő FN-t tartalmazó sorban kapjuk. Ha az FN utasítást a hozzá tartozó DEF FN utasítás előtt hívjuk, akkor ?UNDEF'D FUNCTION ERROR hibajelzést kapunk.

C-64

4.27

DIMRövidítés: dI

Token: \$86 (134)

Belépési pont: \$B080 (45184)Mód: Mind parancs, mind program-módban használható.

Az utasítás a DIM-et követően megadott nevű, típusu, dimenzióju és méretű tömbelemeknek foglal helyet a memóriában. A tömb neve, típusa tetszőleges lehet, az egyes indexek értéke 0-tól a DIM utasításban megadottig terjedhet, de legfeljebb 32767 lehet.

Példák:

```
10 DIM A$(12),B1$(10),B2$(10),MES$(10)
410 INPUT "N=";N:DIM TOMB(N*N+1)
3000 FOR J=1 TO 3:READ A$:MES$(I)=A$:NEXT J
```

READY.

A DIM utasítás használata egyszerű. A problémát inkább az okozza, hogy nagyobb adatmennyiséget a programjaink által kezelhető módon már nehéz megszervezni. A tömbök dinamikusan definiálhatók; erre példa a fenti 410. sor, ahol TOMB dimenzióját az $N*N + 1$ kifejezés adja meg. Az utolsó példa implicit tömbdefiníciót tartalmaz. Ez azt jelenti, hogy tömbváltozókat a tömbök definiálása nélkül is lehet használni, feltéve, hogy indexeik nem haladják meg az indexhatár kezdeti értékét, amit az interpreter 10-re állít be. Az X/1/$ első kiértékelésekor a memóriában a tömb elhelyezésére is sor kerül. A hatás ekvivalens a DIM X/10/$ utasítás hatásával.

/Feltéve, hogy a indexhatár kezdeti értékét tartalmazó memóriát nem irtuk át/.

A legkisebb index értéke \emptyset lehet. Több számítógép nem teszi lehetővé ennek az indexnek a használatát. Ugyanakkor egy tömb \emptyset .elemét különféle számításokban jól fel lehet használni. Például a

```
10 DIM A(20):FOR X = 1 TO 20 : INPUT N
: A(X) = N : A(0) = A(0) + N : NEXT
```

programrész 20 elemet olvas be, összegüket pedig elhelyezi az A(0) tömbelemben.

A CLR utasítás mind az egyszerű, mind a tömbváltozókat törli. A tömbváltozók külön törlése a következő paranccsal /ami programból is használható/ egyszerűen elvégezhető:

```
POKE 49, PEEK (47) : POKE 50, PEEK (48)
```

A 3. fejezetben részletesen leírtuk, hogy egy tömb helyfoglalása hány byte. A következő BASIC paranccsal egyszerűen kiíratjuk a tömb által foglalt helyet:

```
F = FRE (0) : DIM S$ (10,20) : ? FRE (0)
```

Végrehajtás:

A tömb nevének első karakterét az X regiszter tárolja, majd az indexek kiszámítására kerül sor. Ezek végrehajtása során különféle hibajelzéseket kaphatunk. Ha valamennyi indexhatárt sikerrel kiszámítottunk, akkor kerül sor a tömb valamennyi elemének elhelyezésére. Első lépésként a rutin ellenőrzi van-e elég hely a memóriában. Ha nincs, az ugynevezett 'szemétgyűjtési' algoritmus végrehajtására kerül sor. A memóriában, azon a részen ahol a sztringeket tárolja a BASIC, lehetnek olyan részek, amelyekre már nincs szükség. A következő program végrehajtása során például a B\$ változó eredeti értéke, "ABCDE"

szerepel még a sztring területen habár a B\$-ban levő mutató már nem mutat rá. Ezek a felesleges byte-ok törlése, a többi sztring tömörítése a 'szemétgyűjtő' algoritmus feladata. Ha ezután sincs elég hely a memóriában, a parancs vagy program végrehajtása ?OUT OF MEMORY ERROR hibaüzenettel megszakad.

```
20 B$ = "ABCDE" + CHR$ (0)
30 B$ = "IJKLMN" + CHR$ (13)
```

Hibalehetőségek: Hibát elsősorban az indexnatárok kiértékelése közben kapunk. ?OUT OF MEMORY ERROR hibajelzést kapunk, ha nincs elegendő hely a tömb elhelyezésére. Ugyanazt a tömböt csak egyszer definiálhatjuk egy programban. A második kísérletre ?REDIM' D ARRAY ERROR hibajelzést kapunk. Ugyanezt a hibajelzést kapjuk abban az esetben is, ha implicite módon definiáltuk a tömböt:

```
X$ /O/ = "ABCD" : DIM X$ (22)
```

A leggyakoribb hiba, hogy szintaktikus vagy tervezési hiba miatt egy tömbváltozó indexei nem esnek a megfelelő intervallumba. Ilyenkor ?BAD SUBSCRIPT ERROR hibaüzenetet kapunk.

END

Rövidítés: eN

Token: \$80 (128)

Belépési pont: \$A830 (43056)

Mód: Mind parancs, mind program-módban használható.

Az END végrehajtása a program futásának azonnali befejezését jelenti; az interpreter a READY üzenet kiírásával visszaadja

a vezérlést a szövegszerkesztőnek. A CONT parancs a program futásának folytatását eredményezi.

Szintaxis: END . Az utasításnak nincsenek paraméterei. Vagy a sor utolsó utasítása, vagy kettőspont /:/ követi.

Példák:

```
100 PRINT#1: CLOSE1: GOSUB 200: END
130 IF SAV<>TR THEN PRINT "*** NEM JO SAVSZAM ***":END
131 :
2000 GOSUB 1000: END: GOSUB 2000: END
2001 :
60000 END
```

Négy példánkban az END különböző célu alkalmazásait igyekeztünk bemutatni. Az első példa a nyomtató lezárása után, majd egy rutin meghívása után befejezi a program futását. A második példában az END egy hibaág végén jelzi az interpreternek, hogy a program futását fel kell függeszteni. A harmadik példa nem egy kész program részlete, a program ellenőrzésére három töréspontot is elhelyeztünk. A program megállása, a változók ellenőrzése után a CONT paranccsal folytathatjuk a program futását. Az utolsó példában a 60000 sorszámmal kezdődő alprogramok elé tesz egy END utasítást, nehogy 'rácsorogjon' a program vezérlése az alprogramokra.

Végrehajtás: Az END és a STOP ugyanazt a rutint használja. Egyetlen különbség, hogy mikor az END hívja meg a rutint az átvitel bit /C flag/ alacsony. Ebben az esetben a rutin nem nyomtatja ki a BREAK IN ... üzenetet,

Hibalehetőség: Nincs. /Leszámítva azt az esetet, hogy vezérlési hiba folytán rossz helyen áll meg a program/.

EXP

aritmetikai függvény

Rövidítés: eXToken: \$BD (189)Belépési pont: \$BFED(49133)Mód: Mind parancs, mind program módban használható.

eX kiszámítására szolgál $/e = 2.7182818.../$ x értéke megközelítőleg a $/-88,88/$ intervallumba kell hogy essen.

Szintaxis: EXP (<aritmetikai kifejezés>). Ha a kifejezés értéke nagyobb mint 88, ?OVERFLOW ERROR hibajelzést kapunk.

Példák:

```
10 PRINT EXP(10) : REM = 22026.4658
20 Y=EXP(1) : REM Y=2.7182818..
30 PRINT EXP(LOG(X)) : REM =X
40 FOR N=0 TO 10:P(N)=EXP(-M)/FACT(N)
50 NT=NE*EXP(-B*EXP(-K*T))
```

READY.

Hasonlóan az SQR függvényhez az EXP függvény is a hatványozás speciális esete. $EXP /Q/ = 2.7182818 \uparrow Q$. SQR-hez hasonlóan azonban sok esetben szükség van rá és kényelmesebb így használni.

Első két példánk az EXP használatát mutatja. A következő programsor azt mutatja be, hogy az EXP függvény a LOG függvény inverze, leszámítva természetesen a kerekítési hibákat. Az utolsó két képlet valószínűségi, illetve statisztikai programokban használható. Az első a Poisson-eloszlást számítja ki, felhasználva az előzetesen már definiált FACT /N/ függvényt. Az utolsó képletet, ami egy un. logisztikus növekedési függvényt definiál a népesedési folyamatok modellezésében lehet felhasználni.

C-64

4.32

Magát az e számot sokféleképpen lehet definiálni. Egyik definíció sem elemi, lévén e irracionális szám. e^x -et legegyszerűbben a

$$1 + x + x^2/2! + x^3/3! + \dots$$

hatványsorral lehet kiszámítani. Az e^x függvény deriváltja önmaga.

Végrehajtás: A rutin e^x kiszámításához nem a hatványfüggvényt kiszámító rutint hívja meg, hanem saját hatványsort használ. A számítás lépései a következők:

- /i/ Az argumentum értékét $1/\log_e 2$ -vel megszorozza.
- /ii/ Ellenőrzi, hogy az argumentum a megengedett értékhatárba esik-e.
- /iii/ Az argumentum exponens részét a verembe tölti, a 'maradék' argumentum így 0 és 1 közé esik.
- /iv/ A hatványsor-kiszámító rutin segítségével előállítja $2^{\uparrow(x/\log_e 2)}$. Az exponens részt visszaállítja, így végül is $e^{\uparrow x}$ -et kapunk.

Hibalehetőség: Ha az argumentum értéke túl nagy, akkor **OVERFLOW ERROR** hibajelzést kapunk. Ha az argumentum értéke túl kicsi, akkor hibajelzés nélkül a gépi nullával számol tovább az interpreter.

FOR..TO.. [STEP] ...

Rövidítések: fO, stE

Tokenek: FOR \$81 (129)
TO \$A4 (164)
STEP \$A9 (169)

Mód: Mind parancs, mind program módban használható.

Lehetővé teszi a FOR..TO.. [STEP] .. és a megfelelő NEXT utasi-

tások közti programrész többszöri végrehajtását. A NEXT utasítás végrehajtásakor az interpreter ellenőrzi a változó értékét, megkeresi a hozzá tartozó FOR..TO.. [STEP] .. utasítást, a STEP részben definiált értékkel megnöveli a ciklusváltozó értékét, és ellenőrzi, hogy beleesik-e a TO-ban definiált értéktartományba. Ha igen, a vezérlés a FOR..TO.. [STEP] .. utasítást követő első utasításra adódik át, ha nem, a NEXT utáni első utasításra.

Szintaxis: FOR<valós változó> = <aritmetikai kifejezés> TO <aritmetikai kifejezés> [STEP <aritmetikai kifejezés>].

A FOR utáni változót ciklusváltozónak, az egyenlőséggel utáni kifejezés értékét kezdőértéknek, a TO utáni kifejezés értékét végértéknek, a STEP utáni kifejezés értékét növekménynek hívjuk. Ha a STEP hiányzik, az interpreter a növekményt 1-nek tekinti.

Példák:

```
10 FOR J=1 TO 1000 : PRINT "●":NEXT
20 FOR J=1 TO 1000 : PRINT J:NEXT
30 FOR J=1 TO 1000 : NEXT:
```

READY.

A fenti példák a legegyszerűbb ciklusokat mutatják be. Mindegyik esetben J a ciklusváltozó, melynek értékét a cikluson belül nem változtatjuk. Valahányszor a NEXT utasítás végrehajtásra kerül J értéke 1-gyel nő, ez ugyanis a STEP utasítás hiányában a növekmény értéke. Amikor a ciklusból kilépünk a J értéke 1001.

```
10 FOR I=0 TO 255:POKE 1024+I,I:POKE 55296+I,14:NEXT I
20 FOR I=255 TO 0 STEP -1:POKE 1024+I+1,PEEK(1024+I):NEXT I
```

READY.

Az első program a képernyő-memóriába írja be a számokat 0-tól 255-ig. Ennek hatására a képernyő tetején az összes lehetséges karakter kiírásra kerül. A ciklusban használt K változó mutatja, hogyan lehet biztosítani, hogy a ciklus ismétlődő végrehajtása

során egyes változók értéke más és más legyen.

A második program sor a képernyő-memória tartalmát tolja el egy pozícióval jobbra. Az egyes karakterek másolását hátulról előre kell végrehajtanunk, különben a képernyőt teljes egészében az első karakter töltené meg. Ezért használjuk a STEP-1 utasítást.

A ciklusok használata nem szokott különösebb problémát okozni, mégis néhány megjegyzést teszünk helyes használatukról. Először a szintaxisról. A ciklusváltozónak mindig egyszerű valós változónak kell lennie. `FOR x% = 1 TO 9` és `FOR x /0/ = 0 TO 2 STEP-1` egyaránt helytelen. A ciklus a ciklusváltozó alsó és felső határai-
val is lefut. Például a `FOR I = 0 TO 5` ciklusutasítás `I = 0,1,2,3,4,5` értékekkel hajtódik végre.

Előfordulhat, hogy a ciklusváltozó kezdőértéke már kívül esik azon a határon, ameddig futnia kellene. Például a `FOR x = 2 TO 1: NEXT` esetben. Ilyenkor `x` értéke 1 lesz, és a ciklus egyszer lefut. Ez az eset általában a STEP-1 kihagyásakor fordul elő.

Ha a program a ciklusváltozó és a növekmény értékét pontosan tárolta, akkor a ciklus - néhány extrém esettől eltekintve - annyiszor fut, ahányszor matematikailag futnia kell. Általában az egész változók értékeit és a decimális törteket az interpreter pontosan tárolja. Ennek megfelelően a

```
FOR X = 1 TO 1000    illetve a
FOR X = . 5 TO 1000 . 0625
```

utasítások esetén a ciklusmag 1000-szer illetve 16008-szor kerül végrehajtásra. A

```
FOR X = 1 TO 1000 STEP 1/3
```

utasítással már probléma lehet, célszerű a

```
FOR X = 1 TO 1000.1 STEP 1/3
```

utasítással helyettesíteni.

A BASIC interpreter megengedi a ciklusok egymásba ágyazását is. A következő program-séma illusztrálja ezt a helyzetet.

```
FOR X = X1 TO X2:
  FOR Y = Y1 TO Y2:
    FOR Z = Z1 TO ZR
      ...
    <ciklusmag>
      ...
    NEXT Z
  NEXT Y
NEXT X
```

A FOR utasítás végrehajtásakor 18 byte-nyi információ kerül a verembe. A FOR utasításon kívül a GOSUB is használja a vermet, együttes használatuknak így határt szab a verem nagysága /256 byte/. Az interpreter minden egyes FOR utasítás végrehajtásakor ellenőrzi, hogy ez a ciklusváltozó szerepel-e a veremben. Ha igen, a 'verem teteje' mutató erre a változóra fog mutatni, a verem többi része elvész.

A ciklusok tetszőleges strukturában egymásba ágyazhatók:

```
10 FOR X=XA TO XB:FOR Y=YA TO YB : NEXT Y
20 FOR A=AA TO AB: FOR C=CA TO CB: NEXT C,A,X
```

A DO...UNTIL konstrukció a következő ciklussal érhető el:

```
100 OK=-1:FOR J=KEZD TO 9E9
110 IF NOT OK THEN J=9E9:GOTO 200
125 <UTASITASOK>
150 IF <TESZT> THEN OK=0
175 <UTASITASOK>
200 NEXT J
```

A DO.. WHILE konstrukció a következő ciklussal érhető el:

```
FOR J = -1 TO: ... :J = <teszt> : NEXT
```

Végrehajtás: A NEXT utasításnál részletezzük a ciklusutasítás működését.

Hibalehetőségek: Az aritmetikai kifejezések kiértékelése számos hibát eredményezhet. A ciklusutasítás használatában már tervezési hibák is előfordulhatnak. Ezek közül a leggyakoribbak:

- /i/ A negatív növekmény lemarad
- /ii/ NEXT hiányzik
- /iii/ Az egymásba ágyazott ciklusváltozók cseréje
- /iv/ Ugyanannak a változónak beágyazott ciklusokban való használata
- /v/ Egy RETURN nélküli GOSUB könnyen okozhat
? NEXT WITHOUT FOR hibaüzenetet
- /vi/ Ugyanezt a hibaüzenetet kapjuk abban az esetben, ha
a NEXT utasításban egy nem létező ciklusváltozót
használunk.
- /vii/ Ha egész sztring vagy tömbváltozót használunk ciklus-
változónak ?SYNTAX ERROR hibaüzenetet kapunk.

FRE

aritmetikai függvény

Rövidítés: FR

Token: \$A7 (167)

Belépési pont: \$B37D (45949)

Mód: Mind parancs, mind program módban használható.

Kiszámítja hány szabad byte található a BASIC munkaterületen, azaz mennyi a különbség a tömbök vége és a sztringek eleje közt. Ezt megelőzően azonban először a sztringeket egyetlen összefüggő tömbbe tömöríti.

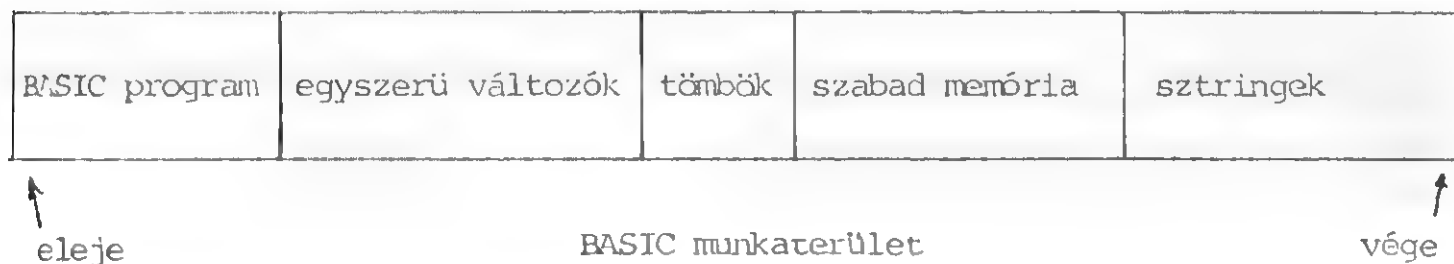
Szintaxis: FRE (<kifejezés>). FRE alakját tekintve függvény, de az argumentum kiszámítására vagy ellenőrzésére nem kerül sor. FRE valójában egy nullaváltozós függvény. Általában PRINT FRE (\emptyset) vagy F = FRE (\emptyset) alakban használjuk. Az FRE (X%), FRE(A + X%), FRE(X) kifejezések szintaktikusan helyesek.

Példák:

```
10 PRINT "SZABAD BYTE-OK SZAMA=";FRE(0)
20 X=FRE(0):DIM X%(278):PRINT FRE(0)-X
```

Az első példa egyszerűen kiírja a szabad memóriakapacitást. A második esetben a FRE függvényt annak kiszámítására használjuk, hogy egy sztring tömb definiálása a memóriában hány byte-ot foglalt le.

A BASIC munkaterületen a program egyes részei a következőképpen helyezkednek el:



Az alábbi példa egy beolvasó rutin, amely 20 egymás utáni karaktert olvas be a billentyűzetről:

```

10  FOR I = 1 TO 20
20  GET X$: IF X$ = "" GOTO 20
30  IS = IS + X$ : NEXT I

```

X\$ minden esetben 1 hosszúságú és I\$ minden egyes kiszámítása I\$ számára a sztringek közt újabb helyet foglal le. A 30. sor

végrehajtásakor a memória $n(n+3)/2$ byte-ját foglalják le X\$ és I\$ előző értékei. A 20 karakter beolvasása a memóriában 230 byte-ot foglal el.

Nagy tömbök /például DIMX\$ (512) / nagymértékben lassítják a 'szemétgyűjtési' eljárást. Ezen csak akkor lehetne segíteni, ha a sztring területen levő sztringek maguk is tartalmaznának mutatókat, amelyek megmondanák, melyik tömb, melyik eleméről van szó.

GET és GET

input utasítás

Rövidítés: gE és gE #

Token: \$A1 (161)

Belépési pont: \$AB7A (43898)

Mód: Csak program-módban használható.

Az utasításokban megadott input perifériáról egyetlen byte-ot tudunk az utasítás segítségével beolvasni. A billentyűzet esetén, ha egyetlen karakter sincs a billentyűzet-pufferben, akkor az eljárás a null-sztringgel tér vissza.

Az utasítás végrehajtásakor az ST állapotjelző byte értéke 0 lesz. Ha az utasítással egy helyesen lezárt, szalagon vagy lemezen tárolt file végét olvassuk, akkor ST értéke 64 lesz, és a rutin egy CHR\$ 13 Carriage return értékkel tér vissza.

Szintaxis: GET # <aritmetikai kifejezés>, <változólista>

A változólista legalább egy elemet kell, hogy tartalmazzon, elemei általában sztring változók. Az aritmetikai kifejezés az input file logikai file számát definiálja.

Példák:

```

5 GET X$ : IF X$="" THEN GOTO 5
10 PRINT " ";X$;" ";ASC(X$) : GOTO 5
(*) 200 GET A$,B$,C$:PRINT A$+B$+C$: GOTO 200

```

Az első példánk segítségével kipróbálhatjuk, hogy az egyes billentyűket a GET hogyan is olvassa be. Az 5-ös sorszámú utasításban addig várunk, míg legalább egy karakter nem kerül a billentyűzet-pufferbe, azután ezt 'visszairjuk' a képernyőre és kiírjuk ASCII kódját is. Érdekes kipróbálni a [RETURN] [DELETE] [RVSON] stb. billentyűk lenyomásának és kiírásának hatását.

A (*) példában 3 byte-ot olvasunk be egy végtelen ciklusban.

A GET utasítás - szemben az INPUT utasítással - byte-onként olvassa be az adatokat, ezért alkalmas ismeretlen strukturájú rekordok beolvasására is. Ezt legegyszerűbben az alábbi ciklussal tehetjük meg:

```

1000 GET#8,X$: IF ASC(X$)=13 GOTO 3000
1010 IN$=IN$+X$ : REM SZTRING OLVASASA
1020 GOTO 1000

```

Megjegyezzük, hogy 50 GET A szintaktikusan helyes. Ha azonban A értéke nem a 0-9 intervallumba esik, akkor egy ?SYNTAX ERROR hibaüzenetet kapunk. Ha a következő byte ; vagy : , akkor ?EXTRA IGNORED hibaüzenetet kapunk. Célszerűbb a GET A\$ utasítás alkalmazása, majd azt követően annak ellenőrzése, hogy számjegyet olvastunk-e be.

Billentyűzet-puffer A GET utasítás /feltéve, hogy a billentyűzet az elsődleges input eszköz/ az input puffer első karakterét olvassa be. Ezeket a karaktereket a hardver megszakító rutin helyezi el a memóriába. /Másodpercenként kb. 60-szor hajtódik vég-

re./ A billentyűzet-puffer a \$277-\$280 /631-640/ címeken található. A \$C6 /198/ címet használja az interpreter a pufferben tárolt karakterek /byte-ok/ számának tárolására / ≤ 10 /. A puffert 'kiüríthetjük' a POKE 198,0 utasítással, vagy a

```
10 GET X$ : IF X$ > "" THEN 10
```

utasítással. A billentyűzet-puffer léte a következő program végrehajtásával érzékeltethető:

```
10 FOR J=0 TO 3000 : NEXT :  
   FOR J=0 TO 20 : GET X$ : PRINT X$ : NEXT
```

A RUN parancs kiadása után - amíg az első sor fut - billentyűzünk be néhány karaktert. A képernyőn csak az utolsó tízet látjuk, ha ennél többet irtunk be, az első elvésznek.

Szalagos file Abban az esetben, ha a GET utasításban definiált logikai file szám egy kazettás file-ra utal, a karaktert a kazetta-pufferből veszi a GET rutin, nem pedig a billentyűzet-pufferből. A kazetta puffer a \$33C-\$3FB /828-1019/ címeken helyezkedik el. Ha a puffert teljes egészében beolvastuk, a program futását az interpreter felfüggeszti, és a következő rekordot betölti a szalagról, majd a kazetta-puffer mutatóját /\$A6,66/ nullára állítja. A 'szalag vége' jelnek egy nulla byte felel meg. Ennek olvasása az ST állapotjelző byte-ot 64-re állítja. Ha ST értékét nem ellenőrizzük akkor a következő GET# /vagy bármilyen I/O utasítás/ ST értékét nullázza és a kazettáról további adatok olvashatók.

Lemezes file Minden olyan esetben, ha a GET# utasítás hatására ST értéke nullától különböző lesz, a rutin egy CHR\$(13) sztringgel tér vissza a BASIC programba. Nem csak az EOI jel, hanem az időtullépés is módosítja ST értékét, így a lassu perifériák csupa CHR\$(13) jelet küldenek.

C-64

4.41

Végrehajtás: A rutin GET része a \$FFE4 KERNEL rutint használja, amelyik az akkumulátorba helyezi el a megfelelő byte-ot és esetleg ST értékét is állítja. Ezt követi egy 'értékadó' rész, amelyik megegyezik a READ és az INPUT utasítások hasonló részével. Ha az interpreter egy# jelet talál, akkor azt követő aritmetikai kifejezés értékének megfelelő eszköztől olvassa be a következő byte-ot.

Hibalehetőségek: Önmagában a GET legfeljebb szintaktikusan hibás. A GET# utasításban a logikai file-t előbb meg kell nyitnunk. Problémát szokott okozni, ha az ST vizsgálata elmarad, és így esetleg az EOF jel után is tovább olvassuk a file-t.

GO

Rövidítés: nincs

Token: \$CB (203)

Mód: Mind parancs, mind program módban használható.

Belépési pont: nincs

A GO utasítás lehetővé teszi a GOTO utasítás GO TO alakban való írását.

Szintaxis: GO <szóközök> TO <sorszám>.

Végrehajtás: Az interpreter először ellenőrzi a szintaxist, majd végrehajtja a megfelelő GOTO utasítást.

Hibalehetőségek: ugyanazok, mint a GOTO utasításnál.

C-64

4.42

GOSUBRövidítés: goSToken: \$8D (141)Belépési pont: \$A882 (43138)Mód: Mind parancs, mind program-módban használható.

Végrehajtja a GOSUB után megadott sorszámú programban kezdődő alprogramot. Ez azt jelenti, hogy feltétel nélküli vezérlésátadás jön létre a GOSUB utasításban megadott számú sorra. Amikor ezt követően a legközelebbi RETURN utasításhoz ér a program, a vezérlés visszakerül a GOSUB utasítást követő első utasításra.

Szintaxis: GOSUB <sorszám> . A sorszám előjel nélküli egész konstans lehet.

Példák:

```

FOR V=0 TO 24:FOR H=0 TO 39 : GOSUB 600 NEXT H,V
:
100 E=0:FOR I=1 TO LEN(S$)
110 IF MID$(S$,I,1)=":" THEN E=E+1
115 NEXT
120 GOSUB 200:GOSUB 6000
200 IF E=0 THEN A$=UZEN$(0)
210 IF E>0 THEN A$=UZEN$(1)
220 PRINT" ";A$: FOR I=1 TO 2000:NEXT:PRINT" "
: RETURN
5000 GOSUB 5010
5010 REM *** CSIPO00 ***

```

Első példánk azt mutatja, hogy használható a GOSUB utasítás parancs-módban. Az 1000 sorszámú szubrutin a kurzor helyét

változtatja a képernyőn. A parancs ellenőrzi a rutin helyességét minden lehetséges értékre.

A második példa egy ellenőrző rutin része. Megvizsgálja, hogy az inputként beolvasott S\$ sztring tartalmaz-e : jelet. A vizsgálat eredményétől függően állítja be UZEN\$ tartalmát. A hozzá tartozó alprogram pedig kiírja a képernyőre az üzenetet, majd meghív egy másik - a példában nem részletezett rutint - amelyik kiírja az Operátor tennivalóját.

A fenti utolsó példa egy olyan programrészt mutat be, aminek két belépési pontja van: GOSUB 510 egyszer, GOSUB 500 kétszer 'csipog'.

Még abban az esetben is, ha a szóban forgó programrész csak egyszer kerül felhasználásra a programban előfordulhat, hogy célszerű alprogramként kidolgozni. Ilyenek például a következő programrész alprogramjai:

```
7000 IF S$="S" THEN GOSUB 5000: GOTO 6000
7010 IF S$="L" THEN GOSUB 5100: GOTO 6000
7020 IF S$="Q" THEN GOSUB 5200: GOTO 6000
7030 IF S$="$" THEN GOSUB 5300: GOTO 6000
7040 GOTO 6000
```

Általában, ha egy programrész hosszabb, mint egy sor, vagy többszörösen egymásba ágyazott IF utasításokat tartalmaz, célszerű megfelelően kialakított alprogramokat használni.

A GOSUB utasítások egymásba ágyazhatóak. Ennek azonban határt szab - nem csak a program áttekinthetősége -, hanem a verem nagysága is. Ilyen esetekben ügyelnünk kell arra, hogy a RETURN utasítás mindig csak a legutolsó GOSUB után tér vissza, s ezért a vezérlést pontosan meg kell terveznünk. Példa erre a következő program.

C-64

4.44

```

10 GOSUB 5000
   <SZAMITASOK>
   . . .

5000 <SZAMITASOK>
   GOSUB 10000
   IF E=0 THEN RETURN
   . . .
   E=1
   RETURN

10000 <SZAMITASOK>
   IF <TESZT> THEN E=1
   RETURN

```

Az 5000 számu sorban kezdődő alprogram egy további alprogramot /10000/ hív meg. Ez az alprogram hiba esetén az E értékét 0-ra állítja. Ebben az esetben a vezérlés azonnal visszatér a 10. sor alá. Mivel az 5000 alatti alprogram a sikeres visszatérés esetén maga is beállítja E értékét - ezt a 20-as sorral kezdődő programrészben is felhasználhatjuk.

Végrehajtás: Az interpreter ellenőrzi, van-e elég hely a veremben /összesen 6 byte-nyi helyre van szükség/. Ha nincs elég hely, a program végrehajtása ? OUT OF MEMORY ERROR hibaüzenettel befejeződik. Ha az ellenőrzés sikeres 5 byte-nyi információ kerül a verembe. A CHRGET rutin mutatója, az aktuális program-sor sorszáma, és a GOSUB token /\$8D/. Ezt követően a GOSUB utasítás ugyanugy hajtódik végre, mint a GOTO utasítás. 2 byte-os egész számként előállítja a megfelelő sorszámot, ennek értékétől függően vagy a program elejétől, vagy az aktuális sortól elkezd a sor keresését. Ezután az interpreter a BASIC utasítások végrehajtását ellenőrző ciklus elejére adja át a vezérlést.

Hibalehetőségek: Ha a GOSUB utasításban megadott számú program-sor nem létezik, akkor ?UNDEF'D STATEMENT ERROR hibajelzést kapunk. A sorszám előállítása az első nem numerikus karakterig történik. Így például GOSUB 1210 hibajelzés nélkül végrehajtodik, és ekvivalens a GOSUB 12 utasítással. Mint fent is jeleztük, amennyiben a veremben már nincs elég hely ? OUT OF MEMORY ERROR hibaüzenetet kapunk. 100 GOSUB 100 például mindig ezt a hibajelzést adja, szemben a 100 GOTO 100 programrésszel, ami hibajelzés nélküli végtelen ciklust eredményez.

GOTO /és GO TO/

Rövidítés: gO /=goto/

Token: \$89 (137)

Belépési pont: \$A89F (43167)

Mód: Mind parancs, mind program módban használható.

Feltétel nélküli vezérlésátadás. A program a GOTO utasításban megadott számú sor végrehajtásával folytatódik.

Szintaxis: GOTO <sorszám>

Példák:

```
D$ = "1234" : GOTO 1500
1000 GET X$ : IF X$ = "" GOTO 1000
GO TO 2000
```

Az első példa a GOTO direkt /parancs/ módban való használatát mutatja be. A D\$ beállítása után a program futása az 1500. sor-számu programsorral kezdődik. Az utasítás hatása nem azonos RUN 1500-al, mert az előbb töröl majdnem mindent, például a D\$ értékét is.

A második példa egy rövid ciklust tartalmaz. Amint megnyomunk egy billentyűt, az IF feltétel hamissá válik és a GOTO 100 utasítás helyett a következő sorra kerül a vezérlés.

A harmadik példa egyszerűen illusztrálja a GO TO használatának lehetőségét. A GOTO utasítás végrehajtásához az interpreternek meg kell találnia a megfelelő sorszámú programsort. Ha az `x GOTO y` utasítás feldolgozása folyik, akkor az interpreter összehasonlítja az `x` és `y` sorszámok felső byte-jait. Ha `y` felső byte-ja nagyobb mint `x` felső byte-ja, akkor a megfelelő sor keresése a GOTO utasítástól, különben a program elejétől kezdődik. A `25600 GOTO 0`, `256000 GOTO 25825` programsorok esetén a 0 illetve a 25825 sorszámú sorok keresése a program elejétől kezdődik. A `25600 GOTO 25856` és a `25600 GOTO 43000` utasítások esetén a GOTO-tól kezdődik a megfelelő programsor keresése.

Végrehajtás: Először a sorszám előállítására kerül sor, amit az interpreter 2-byte-os egész számként tárol. A rutin a GOTO utáni karaktereket olvassa, egészen az első nem számjegy karakterig. A fentebb vázolt módon kerül sor a megfelelő programsor megkeresésére. Ha ilyen sor nem létezik, akkor az `?UNDEF'D STATEMENT ERROR` hibaüzenetet kapjuk. Különben a rutin a CHRGET mutatóját az utasításban definiált programsort közvetlenül megelőző 0 byte-ra állítja, majd az interpreter visszatér a BASIC futását ellenőrző ciklus elejére.

Hibalehetőségek: A `<sorszám>` előállítása az első nem numerikus karakterig tart. Az addig előállított sorszámra kísérli meg átadni a vezérlést az interpreter. Például `GOTO 1X` ekvivalens a `GOTO 1` utasítással. Nem létező sorszám esetén `?UNDEF'D STATEMENT` hibajelzést kapunk.

Rosszul tervezett programok esetén könnyen előfordulhat, hogy a vezérlés nem a megfelelő helyre adódik át. Ilyen esetben a program strukturáját újból kell ellenőrizni /majd a szükséges változtatásokat végrehajtani./

C-64

4.47

IFRövidítés: nincs

Token: \$8B (139)

Belépési pont: \$A927 (43303)Mód: mind parancs, mind program-módban használható.

Feltételes vezérlésátadást hoz létre, attól függően, hogy az IF kulcsszót követő feltétel teljesül-e vagy sem. Ha igen, akkor a feltételt követő utasítás kerül végrehajtásra, ha nem, akkor a következő programsorra kerül a vezérlés.

Szintaxis:

$$\text{IF} \left\{ \begin{array}{l} \langle \text{aritmetikai kifejezés} \rangle \\ \langle \text{logikai kifejezés} \rangle \end{array} \right\} \left\{ \begin{array}{l} \text{THEN} \left\{ \begin{array}{l} \langle \text{sorszám} \rangle \\ \langle \text{utasítás} \rangle \end{array} \right\} \\ \text{GOTO} \quad \langle \text{sorszám} \rangle \end{array} \right\}$$

$\langle \text{sorszám} \rangle$ csak előjel nélküli egész konstans lehet !

Példák:

```
FOR I=1 TO 1000 : X=RND(1) : GOSUB 100: IF T="*" THEN NEXT
100 IF P=72 THEN P=0: GOSUB 1200
200 IF X=1 THEN IF A=0 AND B=0 THEN GOSUB 300
600 IF 8 AND 7 THEN IDE NEM KERUL A VEZERLES!!!
```

READY.

Az első példa, amelyik a 100-zal kezdődő alprogramot ellenőrzi, példa arra, hogyan lehet az IF utasítást parancs-módban is használni. A 100-as programsor ellenőrzi, hogy P értéke elér-e egy bizonyos határt /72/, s ha igen, akkor 0-ra változtatja s bizonyos korrekciós eljárást hajt végre. A következő példánk azt mutatja, hogy az IF utasítás feltétele tetszőleges kifejezés lehet. Az utolsó programsor egy hibás program része, az IF utáni utasítás sohasem hajtódik végre.

Végrehajtás: A rutin először kiszámítja az IF utáni kifejezést, majd ellenőrzi GOTO vagy THEN következik-e. Ha valamelyiket is megtalálta, kerül csak sor a feltétel eldöntésére; hogy igaz-e vagy hamis. Ha hamis /ami azt jelenti, hogy az első lebegőpontos akkumulátor exponense nulla/ a következő sorra kerül a vezérlés. Ha igaz /az exponens nem nulla/, megvizsgálja a következő karaktert. Ha szám, akkor a GOTO végrehajtását hívja ha nem, akkor végrehajtja a következő utasítást.

Hibalehetőségek: A kifejezés kiértékelése közben számos hibajelzést kaphatunk. A GO TO alak nem megengedett; IF X <> Ø GO TO lØ tehát szintaktikus hibát eredményez. Ha a GOTO egy nem létező sorra mutat, akkor ?UNDEF'D STATEMENT ERROR hibajelzést kapunk. Ha aritmetikai kifejezést használunk a feltétel helyén, akkor csak a Ø számot hamisnak. IF X THEN... és IF X <> Ø THEN... tehát ekvivalensek.

INPUT

Rövidítés: nincs

Token: \$85 (133)

Belépési pont: \$ABBE (43966)

Mód: csak program módban használhatjuk.

Az utasítás lehetővé teszi, hogy a billentyűzetről közvetlenül adatokat adjunk át egy futó BASIC programnak. Az INPUT utasítás a billentyűzetről bevitt értékeket megjeleníti a képernyőn. Az eljárás a RETURN megnyomásával fejeződik be.

Szintaxis: INPUT ["<szöveg>" ;] <változólista>

A <szöveg> tetszőleges karaktersorozat lehet; a <változólista> elemei egymástól vesszővel elválasztott egyszerű vagy tömbváltozók. Az opcionális <szöveg> karaktersorozat, majd azt követően egy kérdőjel íródik a képernyőre, végül ezt követi a villogó

kurzor. A bebillentyűzött értékek feldolgozása a következőképpen történik:

- /i/ Az alfanumerikus karaktereket az interpreter az input-változók részének tekinti, bizonyos jeleket azonban ettől eltérően értelmez. Ilyenek elsősorban a ", : jelek. A " jel utasítja az interpretert, hogy az azt követő jeleket egy sztring részének tekintse. Ez azt jelenti, hogy a vezérlő karakterekkel nem szerkeszthetjük tovább az input sort, hanem a megfelelő invert karakterek íródnak az input sorba. A [RETURN] utasítás mindig befejezi az input sor bevitelét.
- /ii/ Az INPUT utasítást végrehajtó rutinnak közös részei vannak a GET és a DATA utasítások hasonló részeivel, s így a vesszőt /,/ és a kettőspontot /:/ elválasztó jelként értelmezi. ?EXTRA IGNORED hibajelzést kapunk, ha a illeténél több input adatot talált az interpreter a bevitt sorban.
- /iii/ Az INPUT utasítás az input sort a <szöveg> és a kérdőjel /?/ kinyomtatásától a sor végéig /maximum 79 karakter/ dolgozza fel, ezért grafikus képernyőn igen nehézkes az utasítás használata.

Példák:

```
100 INPUT "HONAP"; H$
200 INPUT "CIM /VESSZOT NEM HASZNALHAT! /"; C$
300 INPUT X,Y : REM KEZDOPONT
```

Fenti példák az INPUT utasítás legegyszerűbb használatát mutatják; használatuk nem okozhat problémát. Helytelen adatbevitel esetén azonban furcsa dolgok történhetnek. [HOME] a képernyő bal felső sarkába állítja a kurzort. [RETURN] azonnali terminálást eredményez, [SHIFT-STOP] elkezd betölteni egy új programot, stb.

C-64

4.50

```

1000 INPUT AA$ BB,C% : REM KEZDOERTEKEK
2000 FOR J=0 TO 10 : INPUT X$ (J) : NEXT
2010 FOR J=0 TO 10 : PRINT X$ (J) : NEXT

```

Az 1000 sorszámú utasítás három inputváltozót tartalmaz. A

KUTYA, 56.83, 7.1 [RETURN]

helyes válasz. A program futása az AA\$=KUTYA,BB=56.83, C%=7 értékekkel folytatódik. /Az egész változóra vonatkozó értékadás a szokásos kerekítési szabályok segítségével történik./ A következő válasz, egy ?EXTRA IGNORED üzenetet eredményez:

KUTYA,18,56.83,7.1 [RETURN]

A

KUTYA [RETURN]

válasz a következő sor elején két kérdőjel /??/ és a kurzor kinyomtatását eredményezi; jelezve, hogy további értékeket kell bevinnünk a gépbe.

Következő példáink azt mutatják, hogyan lehet az INPUT utasítást néhány egyszerű trükkel biztonságosabbá tenni.

```

10 INPUT "#####";X$: PRINT X$
20 INPUT "HELLO####";X$ : PRINT X$
30 INPUT "###.#####";X$: PRINT X$
40 INPUT"ALFA= 2 #####";A :PRINT A
100 INPUT "BEVITEL ###";A$
200 POKE 198,3: POKE 631,34:POKE 632,34:POKE 633,20:
210 INPUT X$: PRINT X$

```

READY.

Az első négy sor azt példázza, hogyan lehet a <szöveg> -ben szereplő vezérlő karakterekkel érdekes hatásokat elérni. A 100. sorban található INPUT utasítás elejét veszi az üres in-

putsor bevitelének /ami a [RETURN] azonnali megnyomását jelenti/. Ettől ugyanis a rendszer fejre áll. A [USPC]=[SHIFT SZOKOZ] billentyűk lenyomása a képernyőn nem okoz látható változást, de az interpreter nem szóköznék értelmezi.

A 200-210-es sorokban található rutin a billentyűzet pufferbe három karaktert /"" [DEL] / helyez el. A ? kiírása után a három karaktert beolvassa. Ez azt eredményezi, hogy az input sor már nem lehet üres, és a második " utasítja az interpretert, hogy sztringnek tekintse az inputot. Így az vesszőt /,/ és kettőspontot /:/ is tartalmazhat. A két utasítás valójában egy INPUTLINE X\$ utasítást szimulál.

Végrehajtás: Ez a meglehetősen hosszú rutin párhuzamosan dolgozza fel az input sort és az INPUT utasításban levő változólistát; s az inputsorból előállított értékeket megkísérli hozzárendelni a soron következő input változóhoz. Ha a típus nem egyezik, akkor ?REDO FROM START üzenettel az interpreter újra végrehajtja az INPUT utasítást. Ha az inputsor kevesebb értéket tartalmaz, mint ahány INPUT változó van, akkor a következő sor elejére két kérdőjel /?/?/ íródik ki, és a rendszer a hiányzó értékek bevitelét várja. Ha az inputsor több értéket tartalmaz, mint ahány input változó van, akkor ?EXTRA IGNORED üzenetet kapunk és a program futása folytatódik.

Abban az esetben, ha CMD utasítást hajtottunk végre, akkor a <szöveg> a CMD-ben definiált file-ba íródik ki, a kurzor a képernyőn jelenik meg. Az utasítás azért nem használható direkt módon, mert a végrehajtás igénybeveszi az input puffert.

Hibalehetőségek: A példák ismertetése során már felsoroltuk a lehetséges hibákat. Abban az esetben, amikor a képernyőn szerkesztett bizonylatba írunk be adatokat az INPUT utasítást nem használhatjuk; hibás adatbevitel elrontja a képernyőt.

C-64

4.52

INPUT #Rövidítés: iNToken: \$84 (132)Belépési pont: \$ABA4 (43940)Mód: Csak program módban használható.

Megkönnyíti a háttértárakon rögzített adatok visszaolvasását.

Szintaxis: INPUT # <aritmetikai kifejezés>, <változólista> .
Az aritmetikai kifejezés értékének az 1-255 intervallumba kell esnie, és annak a file-nak a sorszáma, amelyikből az adatokat be kívánjuk olvasni. A <változólista> egymástól vesszővel elválasztott egyszerű- és tömbváltozókból áll.

A file-ből beolvasott karaktereket az interpreter a következő szabályok szerint dolgozza fel:

Az alfanumerikus jelek a beolvasott értékek részét képezik, hasonlóan, mint az INPUT utasítás esetében. A [RETURN] /azaz CHR\$(13)/ olvasásának ugyanaz a hatása, mint a [RETURN] billentyűnek az INPUT utasítás esetén. Hasonlóan a vessző /,/ és a kettőspont /:/ az egyes tételek végét jelzik /kivéve " után/. Az ?EXTRA IGNORED hibajelzéshez hasonló üzenetet nem kapunk és általában egyetlen adat sem vész el. Az input-puffer határt szab az egyetlen INPUT# utasítással beolvasható értékeknek; ezek összesen 79 byte-ból állhatnak.

Példák:

```
10 OPEN 4:1,1
20 FOR J=1 TO 10 : INPUT X$: PRINT#10,X$:NEXT
30 CLOSE 4
40 OPEN 5:1,0
50 FOR J=1 TO 10 : INPUT#5,X$: PRINT X$: NEXT
60 CLOSE 5
```

A fenti példa egy ki-beviteli utasításpár, amelyen nem tüntettük fel a szalag visszacsévévelési, az ST ellenőrző

C-64

4.53

részeket. A file számok természetesen tetszőlegesek lehetnek, de hogy jobban látszódjék, hogy különböző célra használjuk őket, nem egyeznek meg az egységszámmal.

```
10 OPEN 1,0 : REM FILE#1 = A BILLENTYUZET
20 OPEN 3,3 : REM FILE#3 = A KEPERNYO
30 INPUT#1,X$ : PRINT#3,X$ : GOTO 30
```

READY.

A következő példánk azt mutatja, hogyan használhatjuk a billentyűzetet, mint egy file-t. A [RETURN] azonnali megnyomása most nem állítja fejre a programot, mert nem az INPUT utasítást használjuk.

```
5 OPEN 1,8,2,"@:PROBA,S,W"
10 FOR J=1 TO 10 : PRINT#1,STR$(J):NEXT
20 CLOSE1
30 OPEN 2,8,3,"PROBA,S,R"
40 FOR J=1 TO 10 : INPUT#2,X$:PRINT X$:NEXT
50 CLOSE2
```

READY.

Az utolsó programrész-pár egy-lemezes file-ba való írást, majd ugyanennek a file-nak a visszaolvasását mutatja be.

Végrehajtás: Az INPUT és INPUT# utasítások a billentyűzetről illetve a megfelelő file-ból kapott karaktereket az input-pufferbe /\$0200/ másolják. A pufferba való másolást egy RETURN illetve CHR\$(B) olvasása fejezi be. A RETURN helyett azonban egy CHR\$(O) kerül az input-pufferbe. A \$01FF címen is egy vessző található, ez lehetővé teszi, hogy az inputsor első tételét az interpreter ugyanugy dolgozza fel, mint a többi. Az INPUT-ről szóló rész egyik példáját az interpreter így tárolja:

\$1FF	\$200	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	O	1	2
,	K	U	T	Y	A	,	1	8	,	5	6	.	8	3	,	7	.	1	null

A tételek feldolgozása hasonló mint az INPUT és a READ utasítások esetén.

Hibalehetőségek: A szintaktikus hibákon kívül leggyakoribb, hogy a file végén túl akarunk olvasni, vagy a rekordokat nem megfelelően tagolva olvassuk vissza.

INT

Rövidítés: nincs

Token: \$B5 (181)

Belépési pont: \$BCCC (48332)

Mód: Mind parancs , mind program módban használható

Az argumentum egész részét számítja ki.

Szintaxis: INT aritmetikai kifejezés

Példák:

```
10 PRINT INT(X+.5)
20 PRINT INT(123456.789) : REM = 123456
30 PRINT INT(-123456.789): REM =-123457
40 IF X<>INT(X) THEN GOTO 35
```

READY.

A legtöbb kerekítési eljárás az INT függvényt használja. Az első és a negyedik sor egyszerű példákat mutat a kerekítésre. Az első példa azt használja fel hogy 0.000-0.999 egész része egyaránt 0. Ha egy számot a legközelebbi egész számra akarunk felkerekíteni, akkor .5-et kell hozzáadni és utána venni az egész részét.

A második és harmadik példa azt illusztrálja, mi a különbség a pozitív és a negatív számok egész részének képzése között. Utolsó példánk azt ellenőrzi, hogy egész számot adtunk-e be inputként.

Végrehajtás: Az argumentum értéke az első lebegőpontos akkumulátorba kerül, ezután sor kerül egy rutin végrehajtására, amelyik 4 byte-os egész számmá alakítja. Végül ezt a számot a rutin visszakonvertálja lebegőpontos számmá. Ez azt jelenti, hogy az `X=INT (1234567.8)` értékadás helyes, míg az `X%=INT (1234567.8)` hibát eredményez. /Tulcsordul./

Hibalehetőségek: Az argumentum kiértékelése közben számos hibadódhat. Magának az INT-nek a kiszámítása nem okoz hibát.

LEFT\$

Rövidítés: leF /a \$-ral együtt !/ Token: \$C8 (200)

Belépési pont: \$B700 (46848)

Mód: Mind program, mind parancs módban használható.

LEFT\$ kétváltozós sztringfüggvény, amelyik az első argumentumként megadott sztring első karaktereiből egy új sztringet képez. A karakterek számát, amit ehhez felhasználunk a második argumentum definiálja.

Szintaxis: LEFT\$ (<sztring> , <aritmetikai kifejezés>).

Példák:

```
PRINT LEFT$("HOGY VAGY",4) : REM ="HOGY
PRINT LEFT$("HOGY VAGY",123) : REM NEM VALTOZIK
PRINT LEFT$(X$+"                ",15)
PRINT LEFT$(STR$(L)+"          ",15):
PRINT L;LEFT$("                ",15-LEN(STR$(L)))
```

Az első két parancs a LEFT\$ hatását mutatja be. A második példa mutatja, hogy ha az aritmetikai kifejezés meghaladja a sztring hosszát, akkor eredményül az eredeti sztringet kapjuk minden hibajelzés nélkül. Az utolsó három sor példa arra, hogyan egészíthetünk ki egy sztringet szóközökkel éppen 15 karakterre.

LEFT\$ (X\$,N) mindig helyettesíthető a MID\$ (X\$,1,N) kifejezéssel.

Végrehajtás: A paraméterek kiértékelése után, mondjuk a LEFT\$ (X\$,N) végrehajtása közben, az interpreter összehasonlitja N-et és LEN (X\$)-t és a kisebbel számol. Először egy \emptyset byte kerül a verembe, majd az eredmény sztring hossza, végül egy rutin, amelyik közös a LEFT\$, RIGHT\$, MID\$ esetében a sztringet magát is előállítja.

Hibalehetőségek: A paraméterek kiértékelésén túl az $N=\emptyset$ érték is hibát okoz.

LEN

Rövidítés: nincs

Token: \$C3 (195)

Belépési pont: \$B77C(46972)

Mód: Mind parancs, mind program módban használható.

A LEN aritmetikai függvény az argumentum sztring hosszát adja meg.

Szintaxis: LEN (<sztring kifejezés >).

Példák:

```

10 PRINT LEN("HAHO OCSI!") : REM =10
20 X$="ELJEN" : PRINT LEN(X$+"MAJUS")+2
21 :
312 FOR J=1 TO 10
314 PRINT SPC(19-LEN(UZENET$(J))/2);UZENET$(J)
316 NEXT J
499 :
500 IF LEN(BE$)<>13 THEN PRINT "## HIBA ##":GOTO 344
779 :
780 X$="*##!%10"
790 FOR J=1 TO LEN(X$)
900 IF G$=MID$(X$,J,1) THEN RETURN
810 NEXT: PRINT "## FELISMERHETETLEN ##"
```


Az első két példa a függvény hatását illusztrálja. A következő néhány soros rutin az UZENET\$(J) sztringeket írja ki a képernyő közepére. Harmadik példánk egy input ellenőrző rutin, amelyik ellenőrzi, hogy a bevitt sztring az előírt hosszúságú-e. Utolsó példánkban egy egyszerű programot írtunk, amelyik ellenőrzi, hogy G\$ egy előre megadott karakterkészlet eleme-e vagy sem. A ciklus végértékének természetesen 5-öt is írhattunk volna, de a program módosítása ekkor lényegesen nehezebb lenne.

Végrehajtás: A rutin először előállítja az argumentum sztringet, majd hosszát az A és Y regiszterekbe tölti. Ezután egy lebegőpontos konverziós rutin kerül végrehajtásra, amelyik a sztring hosszát lebegőpontos alakban állítja elő.

Hibalehetőségek: A sztring-kifejezés kiértékelése után egy legfeljebb 255 hosszú sztringet kell kapnunk, ebben az esetben LEN hibajelzés nélkül végrehajtódik.

LET

Rövidítés: !E /vagy semmi/

Token: \$88 (136)

Belépési pont: \$A9A4 (43428)

Mód: -Mind parancs, mind program módban használható.

Értékadó utasítás.

Szintaxis:

$$[LET] \left\{ \begin{array}{l} \langle \text{egész változó} \rangle = \langle \text{aritmetikai kifejezés} \rangle \\ \langle \text{valós változó} \rangle = \langle \text{aritmetikai kifejezés} \rangle \\ \langle \text{sztring változó} \rangle = \langle \text{sztring kifejezés} \rangle \end{array} \right.$$

A változók egyszerű és tömb változók egyaránt lehetnek. A LET szó nem kötelező. Ha egy utasítás első karaktere nem token, akkor az interpreter LET-et tételez fel.

Példák:

```
LET X=1234:LET X$="QWE":LET X%=12.45
20 FOR J=1 TO 10: READ X% : LET A%(J)=X%: NEXT
30 IF A+B>C THEN D%=A/B

READY.
```

A parancs módban kiadott utasítások a különböző típusu változók használatát mutatják. Egész változó esetén az aritmetikai kifejezés értékének egész része kerül átadásra. A 2. példában egy összetett értékadás szerepel, egy ciklus a DATA sorokból olvas be, majd ezek értékét egy sztring-mátrixba írja. Utolsó példánk egy feltételes értékadást mutat be.

Néhány mikrogép esetén kötelező a LET használata /ilyen például a ZX81/.

A változók értékei akárhányszor és a program bármely részéből megváltoztathatók. A BASIC nem ismeri a 'lokális' és a 'globális' változók fogalmát. Ez különösen az alprogramok megtervezésekor okozhat problémát. Ha egy rutint a program több részéből is meghívunk, akkor változóit más célokra nem célszerű használni.

A C-64 BASIC a sztringeket kétféleképpen tárolja. A két tárolási mód közti különbség néha fontos lehet. Tekintsük a következő utasítást:

```
10 A$ = "HELLO" : B$ = "HELLO" + " "
```

Az A\$ illetve B\$ értékét a program eltérően tárolja:



A változók területén a sztring neve, hossza, illetve a sztring első elemére mutató 2 byte kerül tárolásra. Amennyiben egy sztring-konstans kerül a változóba, a mutató magába a program-

ba mutat vissza /helykimélés céljából/. A sztringkifejezések eredményeiként előálló sztringek a BASIC munkaterület végén tárolódnak. Ha most egy új programot töltünk be a PROGRAM-ból, akkor A\$ értéke elvész, míg B\$ értéke megmarad.

Végrehajtás: Az interpreter először a változó szintaxisát ellenőrzi, majd megnézi, szerepel-e a memóriában. Ha igen, ellenőrzi az = jelet, kiértékeli az azt követő kifejezést és értéket - a típusától függő módon - átadja a változónak. Amennyiben a változót nem találja, először elhelyezi a memóriában. Ez alkalom adtán elég soká tarthat, hiszen egy egyszerű változó elhelyezéséhez először az összes tömböt el kell tolnia. A rutin a változó típusát a verembe tölti, s a jobb oldalon álló kifejezés kiértékelése után ellenőrzi csak, hogy megegyeznek-e a típusok. A valós és egész változók, kifejezések közt az interpreter automatikus konverziót hajt végre. A rutin elején külön rész ellenőrzi a TI\$-ra vonatkozó értékadást.

Hibalehetőségek: Amennyiben a változó és a kifejezés típusa nem felel meg egymásnak, ?TYPE MISMATCH hibajelzést kapunk. Ha a változó, amelynek értékét a kifejezés definiálja tömbváltozó és indexei nem esnek a megfelelő értékhatárok közé, akkor BAD SUBSCRIPT hibajelzést kapunk. A kifejezés kiértékelése közben további hibaüzeneteket is kaphatunk /DIVISION BY ZERO stb./.

LIST

Rövidítés: LI

Token: \$9B (155)

Belépési pont: \$A69B (42651)

Mód: Mind program, mind parancs módban használható.

A BASIC munkaterületen tárolt programot, illetve annak a LIST paramétereiben specifikált részét írja ki a képernyőre /pontosabban az elsődleges outputra/.

Szintaxis: LIST <sorszám> [- <sorszám>] vagy LIST[<sorszám> -]

Példák:

```
LIST          : REM AZ EGESZ PROGRAMOT KILISTAZZA
LIST 5        : CSAK AZ 5. SORT IRJA KI
LIST 5-135    : AZ 5 ES 135 KOZE ESO SOROKAT IRJA KI
REM          : BELEERTVE 5-T ES 135-T IS
LIST -135     : REM A 135. SORIG LISTAZ
LIST 135-     : REM A 135. SORTOL LISTAZ
10 PRINT "*** HIBA *** " : LIST 456
```

A példák a LIST paramétereinek használatát mutatják be. Normál körülmények között a programlista a képernyőn jelenik meg. Ha azonban az elsődleges outputot a CMD-vel megváltoztattuk, akkor a programlista ott jelenik meg. Az

```
OPEN 3,4 : CMD 3 : LIST
```

parancs a programot a sornyomtatóra listázza ki. A program kassettára vagy lemezre is kilistázható. /Ekkor azonban LOAD-dal már nem tölthető vissza!/

Az utolsó példa mutatja, hogy a LIST program módban is használható; hatása ugyanaz mint a STOP utasításnak, azzal a különbséggel, hogy a program futása a CONT-tal nem folytatható és a LIST paramétereinek megfelelő programsorokat a program kilistázza.

A program listája nem azonos azzal, ahogyan beirtuk az egyes programsorokat. A ?X utasítás a listán PRINT X alakban jelenik meg. A REM alapszó utáni emelt /siftelt/ karaktereket a listázó rutin tokeneknek tekinti és a nekik megfelelő alapszavakat írja ki. A megfelelő memóriahelyek tartalmának módosításával elérhetjük, hogy a megjegyzés sor képernyő vezérlő karaktereket is tartalmazzon. Ilyenkor a LIST hatására igen furcsa kiírási képet kaphatunk. Hasonló a probléma a sztring konstansokban szereplő vezérlő karakterekkel. Gépeljük be a
 10 ?"GO AWAY" utasítást, listázzuk ki a sort és álljunk a kur-

zorral a második idézőjel fölé, szurjunk be az INS billentyű segítségével nyolc szóközt, majd ezt töltsük fel DEL jelekkel /inverz T-ként jelennek meg/. A listázás eredménye ezután 10 PRINT.

A leghosszabb sor, amit ki tudunk listázni egy öt decimális jegyű sorszámból és 251 RESTORE tokenből áll. /Ezt a sort a szóköses módon nyilván nem tudjuk beírni a programba./

Végrehajtás: A rutin számos \emptyset . lapon levő címet használ, ez az egyik oka, amiért listázás után a CONT nem használható. A paraméterek kiértékelése után egy kettős ciklus írja ki a sorokat; a külső ciklus ellenőrzi a sor illetve program végét jelző byte-okat, állítja elő a sorszámot kiiratható alakban. A belső ciklus egy sor kiírását végzi, ez cseréli az alapszavak tokenjét magukra az alapszavakra.

Hibalehetőségek: Gyakorlatilag nincsenek. A paramétereknek nem kell létező sorszámnak lenniük, a LIST 100-utasítás az első 100-nál nem kisebb számu sortól kezdi a program listázását.

LOAD

Rövidítés: 10

Token: \$93 (147)

Belépési pont: \$E167 (57703)

Mód: Mind program, mind parancs módban használható.

Az utasítás a paramétereiben megadott nevű programot adott egységről és adott módon betölti.

Szintaxis: LOAD <sztring kifejezés>[, <aritmetikai kifejezés>][, <aritmetikai kifejezés>]. Valamennyi paraméter opcionális, hiszen a kazettás egységen a következő file egyértelműen azonosítható. A <sztring kifejezés> a program nevét azonosítja. Az első <aritmetikai kifejezés> az egységszám; ha elmarad, ak-

kor a LOAD mindig a kazettás egységre vonatkozik /egységszáma = 1/. A második <aritmetikai kifejezés> a töltés módját határozza meg /secondary adress/. Kazettás file-ok esetén semmilyen hatást sem gyakorol.

Lemezes file-ok esetén az egységszám /8 vagy 9/ mindig kötelező; ilyenkor a harmadik paraméter jelentése a következő:

- 0 a töltés a BASIC munkaterület elejéről kezdődik;
- 1 a töltés a program-file első két byte-ja által meghatározott címtől kezdődik.

Lemezes file-ok esetén a névben szereplő bizonyos karaktereknek speciális jelentése van /lásd az 5. fejezetet!/. Ha a név **x**-gal végződik, például PROG**x**, akkor az utasítás a katalógusban szereplő és "PROG"-gal kezdődő nevű első file-t tölti be. A file-névben szereplő kérdőjelek /?/ tetszőleges karaktert jelentenek. LOAD "HE??O**x**",8 például az első olyan, legalább öt karakteres nevű file-t tölti be, amely nevének első két karaktere HE, 5. karaktere pedig O.

Példák:

```
LOAD          :REM A SZALAGROL AZ ELSO PROGRAM
LOAD "PROG"    :REM A SZALAGROL A PROG NEVU PROGRAM
LOAD "*",8     :REM LEMEZROL A KATALOGUS ELSO PROGRAMJA
LOAD "AS*",8   :REM LEMEZROL AZ ELSO AS-SZAL KEZDODO PROGRAMOT
:
99 LOAD "PROG",3
45 PRINT "VARJ, TOLTOK!":LOAD "PROG2",3
```

Utolsó két példánk BASIC programból használja a LOAD utasítást. Program módban a LOAD utasítás használata eltér a parancs módban való használatától. A töltés befejezése után a program futása előről kezdődik. Amennyiben BASIC programot töltöttünk be, az új, éppen betöltött program kezd el futni. Ezt a lehetőséget overlay technika kialakítására is felhasználhatjuk. A be-

töltés az eredeti program változóit nem törli, feltéve, hogy a másodszorra betöltött program rövidebb az elsőnél. A sztring konstansokkal végrehajtott értékadások eredményei és a függvény definíciók azonban mindenképp elvesznek.

Végrehajtás: Ha az utasítást parancs módban adtuk ki, a képernyőn információkat kapunk az utasítás végrehajtásáról és/vagy a tennivalóinkról.

Kazettás file esetén a következő információkat kapjuk:

```
LOAD "PROGRAM", 1
PRESS PLAY ON TAPE
OK
SEARCHING FOR PROGRAM
FOUND NEV
FOUND PROGRAM
LOADING PROGRAM
READY.
```

A "PRESS PLAY ON TAPE" üzenet megjelenése után kell a PLAY gombot lenyomnunk. Az interpreter addig keres a szalagon, míg az utasításban specifikált file-t meg nem találja. A keresés és töltés alatt a képernyő üres.

Lemezes file esetén az információ kevesebb:

```
LOAD "PROGRAM", 8

SEARCHING FOR PROGRAM
LOADING PROGRAM
READY.
```

Maga a rutin két részből áll, az egyik a töltést a kazettás egységről, a másik a lemezes egységről végzi el. A rutin beállítja a töltés jelzőt /IO. byte = 0/, ellenőrzi a paramétereit.

Hibalehetőségek: Elsősorban kazettás egységről való töltés esetén hardver hibák is történhetnek. FILE NOT FOUND hibajelzést kapunk, ha a szóban forgó file nem szerepel a lemez katalógusában, illetve a kazettán az end-of-tape jelzőn túl olvasott az interpreter. DEVICE NOT PRESENT hibajelzést kapunk, ha az egység számnak megfelelő egység nincs a C-64-hez csatlakoztatva és bekapcsolva /ez a hibajelzés a kazettás egységgel kapcsolatban nem fordulhat elő/. Gépi kódu programok töltése esetén a harmadik paraméter általában kötelező, s elhagyása hibát okozhat. Egy több gépi kódu programból álló program töltőprogramját mutatjuk be végül; az A értékének változtatása nélkül a program egy végtelen ciklusban mindig csak az első programot töltené be:

```
10 IFA=0 THEN A=1:LOAD"SKOT1",8,1
20 IFA=1 THEN A=2:LOAD"SKOT2",8,1
30 IFA=2 THEN A=3:LOAD"SKOT3",8,1
40 IFA=3 THEN A=4:LOAD"SKOT4",8,1
50 SYS40960
```

LOG

Rövidítés: nincs

Token: \$BC (188)

Belépési pont: \$B9EA (47594)

Mód: Mind parancs, mind program módban használható.

Egyváltozós aritmetikai függvény, amelyik argumentumának c alapu logaritmusa számítja ki. Az EXP függvény inverze.

Szintaxis: LOG (<aritmetikai kifejezés>).

Példák:

```

10 PRINT LOG(10)      :REM = 2.3026
20 PRINT LOG(2.718281):REM = 1
30 PRINT LOG(X)/LOG(2):REM 2 ALAPU LOGARITMUS
40 PRINT LOG(EXP(X))  :REM = X
50 DEF FN LG(X) = LOG(X)/LOG(10):REM 10 ALAPU LOGARITMUS

```

Példáink a LOG függvény használatának legegyszerűbb eseteit mutatják be, magára a függvényre statisztikai, tudományos számítások során van szükség.

Végrehajtás: A rutin az előjel ellenőrzésével kezdődik, csak pozitív számoknak van logaritmus. A LOG kiszámítása hatványsorral történik, amelynek összesen $1/4$ tagja van. Ez a hatványsor valójában a szám 2-es alapu logaritmusát számítja ki, ebből egy konverziós algoritmus segítségével kapjuk meg az e alapu logaritmust.

Hibalehetőségek: Ha az argumentum értéke nem pozitív ILLEGAL QUANTITY ERROR hibajelzést kapunk.

MID\$

Rövidítés: mI /beleértve a \$-t is/ Token: \$CA (202)

Belépési pont: \$B737 (46903)

Mód: Mind parancs, mind program módban használható.

2 vagy 3 változós sztring-függvény, amelyik az első paraméterként megadott sztring bizonyos egymás után következő karakteriből egy új sztringet állít elő.

Szintaxis: MID\$(<sztring kifejezés>, <aritmetikai kifejezés> [, <aritmetikai kifejezés>]). Az aritmetikai kifejezések nem lehetnek 255-nél nagyobbak. A második paraméter a sztring azon

karakterhelyét adja meg, ahonnan az új sztring kezdődik. A harmadik paraméter /ha van/ az új sztring hosszát /az átmásolandó karakterek számát/ adja meg. Ha nincs a paraméterként szereplő sztringben már ennyi karakter, akkor a sztring végéig másolja át a karaktereket. Ha a harmadik paraméter hiányzik a 255-nek felel meg. Legyen

```

      X$ = "KOVACS PETER"
      Pozíció = 123456789012
Ekkor MID$(X$, 3,6) = "VACS P"
      MID$(X$, 8,10) = "PETER" = MID$(X$,8).

```

```

N$=MID$(STR$(N),2)
MO$=MID$("JANFEBMARAPRMAJJUNJULAUGSZEOKTNOVDEC",3*M-2,3)

```

Az első példánk a STR\$(N) első karakterét törli. Ha például N=23, akkor STR\$(N) = " 23", de N\$ = "23". Természetesen N=-23 esetén is N\$ = "23"!

A második példa azt mutatja, hogyan lehet MID\$-t 'tömbként' használni. MO\$ az M. hónap hárombetűs rövidítését tartalmazza.

A LEFT\$ és a RIGHT\$ függvények a MID\$ segítségével kifejezhetők:

```

LEFT$(X$,N) = MID$(X$,1,N)
RIGHT$(X$,N) = MID$(X$,LEN(X$) - N+1)

```

Végrehajtás: A rutin először a paramétereket ellenőrzi. Miután kiszámította az új sztring hosszát és az annak elejére mutató 2 byte-ot, mind a 3 byte-ot betölti a verembe, majd meghívja a LEFT\$ rutint, amelyik a sztringet a további műveletek végzéséhez megfelelő helyre tárolja.

Hibalehetőségek: ?ILLEGAL QUANTITY ERROR hibajelzést kapunk, ha a sztring hossza vagy a paraméterek nagyobbak 255-nél. Hasonló jelzést kapunk, ha az eredmény egy nullsztring lenne.

NEW

Rövidítés: nincs

Token: \$A2(162)

Belépési pont: \$A641 (42561)

Mód: Mind parancs, mind program módban használható.

Törli a memóriában tárolt BASIC programot.

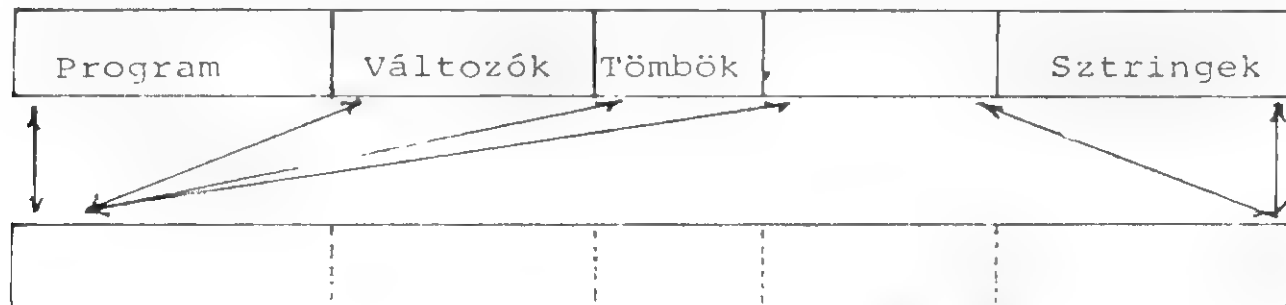
Szintaxis: NEW. A NEW utasításnak nincsenek paraméterei, egy kettőspont /:/ vagy egy programsor végét jelző \emptyset byte követheti.

Példák: NEW

1000 PRINT "ODA AZ EGESZ" : NEW

Mind direkt módban, mind program módban a hatás ugyanaz, a program és a változók törlődnek és a szövegszerkesztő kiírja a READY. üzenetet.

Végrehajtás: Az utasítás valójában nem törli a memória tartalmát, csupán a BASIC munkaterülethez kapcsolódó mutatókat állítja kezdőértékeikre, ahogy az alábbi ábrán is látható:



Az interpreter ezekután úgy tekinti, hogy üres a tár.

Hibalehetőségek: Nincs /de elveszhet a program/.

NEXTRövidítés: nEToken: \$82 (130)Belépési pont: \$AD1D (44317)Mód: Parancs és program módban egyaránt használható.

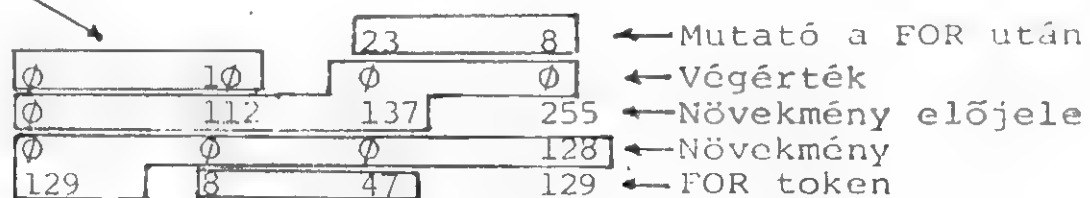
Az utasítás hatására a vezérlés közvetlenül a NEXT-nek megfelelő FOR-t követő első utasításra kerül, feltéve, hogy a STEP-ben specifikált értékkel megnövelt ciklusváltozó még mindig a megengedett értéktartományba esik. Ha nem, a NEXT-et követő első utasítás hajtódik végre.

Szintaxis: NEXT változólista A változólista egymástól vesszővel elválasztott egyszerű, valós változókat tartalmazhat. NEXT I,J és NEXT I : NEXT J egymással ekvivalensek. Amennyiben a NEXT nem tartalmaz további paramétereket, akkor a legutoljára végrehajtott FOR utasítás alá tér vissza a vezérlés. Példákat a FOR utasítás leírásakor adtunk.

A következő rövid program bemutatja, hogyan használja a FOR ... NEXT utasításpár a vermet:

```
1Ø FOR PQ = 512 TO 48Ø STEP - 1
2Ø PRINT PEEK (PQ),
3Ø NEXT
```

A program futása a következőket írja ki a képernyőre:

sorszám

A ciklusváltozó helye

Végrehajtás: A FOR utasítás szintaxisát a FOR utasítás végrehajtásakor ellenőrzi az interpreter. Az összes szükséges pa-

paraméter értéke /összesen 18 byte/ a verembe kerül; legvégén a FOR tokenje: 129. A NEXT utasítás végrehajtása a megfelelő 18 byte megkeresésével kezdődik. Ha a változónak megfelelő részt nem találja meg az interpreter, akkor ? NEXT WITHOUT FOR hibajelzést kapunk. Amikor megtalálta a megfelelő FOR token, megnöveli a STEP-ben specifikált értékkel a ciklusváltozót és ellenőrzi, nem haladta-e meg a felső határt. Negatív STEP esetén az alsó határt ellenőrzi. Ha nem, a CHRGET paraméterét a FOR utánra állítja; ha igen, akkor folytatja a program vagy parancs végrehajtását.

Hibalehetőségek: A NEXT I utasítás törli a veremből a felesleges NEXT-eket. Helytelenül strukturált programok esetén ez ?NEXT WITHOUT FOR hibajelzést eredményezhet.

NOT

Rövidítés: nO

Token: \$A8 (168)

Belépési pont: \$AED3(44755)

Mód: Mind parancs, mind program módban használható.

Egyváltozós logikai és aritmetikai függvény. Az argumentumot nem kell zárójelbe tenni.

Szintaxis: NOT aritmetikai vagy logikai kifejezés

Példák:

```
10 IF PEEK(X)=34 THEN ID= NOT ID
20 PRINT NOT 23456 : REM = -23457
30 IF NOT OL THEN PRINT "!! HIGH !!": END
```

READY.

Az első sor egy listázó rutinban szerepel. Ha a következő byte egy idézőjel /"/, akkor az idézőjel jelenlétét ellenőrző ID változó ellentettjére változik.

A második példa a NOT használatát aritmetikai argumentum esetén mutatja be. $23456 = \$5BA0$, így a NOT műveletet bitenként elvégezve a $\$A45F$ hexadecimális számot kapjuk, amelyik a -23457 számnak felel meg. A nyomtatás eredményeül tehát ennyit kell kapnunk.

Utolsó példánkban a NOT használatát feltételes vezérlő utasításban mutatjuk be. Ha az OK változó nem igaz /azaz 0/, akkor a HIBA szöveg kiíródik a képernyőre és a program megáll.

Végrehajtás: A rutin először az argumentum értékét ellenőrzi, hogy a $-32768-32767$ tartományba esik-e, majd a NOT műveletet bitenként végrehajtja. Ezt követően egy konvertáló rutin a végeredményt újból lebegőpontos számmá alakítja.

Hibalehetőségek: A NOT művelet a logikai műveletek közt a legmagasabb rangú, ezért először a NOT hajtódik végre. Ennek megfelelően tartsuk szem előtt, hogy például a NOT A AND B és /NOT A/ AND B ekvivalens.

ON

Rövidítés: nincs

Token: \$91 (145)

Belépési pont: \$A94A (43338)

Mód: Mind parancs, mind program módban használható.

Többirányú elágazást tesz lehetővé.

Szintaxis:

$$\text{ON } \langle \text{aritmetikai kifejezés} \rangle \left\{ \begin{array}{l} \text{COSUB} \\ \text{GOTO} \end{array} \right\} \langle \text{sorszámlista} \rangle$$

A sorszámlista egymástól vesszővel elválasztott előjel nélküli egész konstansokat tartalmaz. Az aritmetikai kifejezést megelőző ki először az interpreter beépített -2^{31} interpreter

vallumba kell esnie/, majd a lista annyiadik elemének megfelelő programsorra adódik át a program vezérlése. GOSUB esetén a verembe még a visszatérést biztosító információk is bekerülnek.

Példák:

```
1000 ON SGN X +2 GOTO 2000, 3000, 4000
```

```
60 ON 1+RND 1 * 10 GOTO 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000
```

```
200 ON Q GOSUB 100, 200, 300
```

Az első sor egy előjel szerinti elágazást mutat. Ha $X < 0$ a 2000., ha $X = 0$ akkor a 3000., ha pedig végül $X > 0$, akkor a 4000. sorral folytatódik a program. A második sor egy véletlen programelágazást mutat be, ilyen programrész főleg játékprogramokban fordul elő. Az ON utasítás az 'üres' sorszámot is megengedi; ez a nulla értéknek felel meg, s így a program legelső sorát jelenti. Ha X értéke \emptyset vagy 4, nem kapunk hibajelzést, a program futása a következő sorral folytatódik.

Végrehajtás: A rutin először az aritmetikai kifejezést értékeli ki, majd ellenőrzi, hogy 1 byte-os számmá konvertálható-e. Ezt követően egy ciklus végigolvassa a sorszámokat, míg a sor végére nem ér vagy meg nem találja a lista megfelelő elemét. Ezt követően vagy a GOTO vagy a GOSUB utasítás hajtódik végre.

Hibalehetőségek: ?ILLEGAL QUANTITY hibajelzést kapunk, ha az aritmetikai kifejezés egész része nem esik a 0-255 intervallumba. ?SYNTAX ERROR üzenetet kapunk, ha az aritmetikai kifejezést nem GOTO vagy GOSUB token követi. A GO TO használata nem megengedett! Végül ha a szóban forgó számú sor nem létezik, akkor ?UNDEF'D STATEMENT hibajelzést kapunk.

OPENRövidítés: OPToken: \$9F (159)Belépési pont: \$E1BD 57789Mód: Mind parancs, mind program módban használható.

Az utasítás első paramétereként szereplő számot, mint logikai file számot felveszi egy táblázatba az esetleges egységsszámmal, illetve megnyitási móddal együtt. Amikor egy BASIC utasítás, például egy PRINT# erre a logikai file számra hivatkozik, a hozzá tartozó egységsszám és megnyitási mód paramétereket a táblázatból előkeresi az interpreter, és ezek értékőnek megfelelően hajtja végre az illető utasítást. Az OPEN utasítás hatására azoknál az egységeknél, ahol ez szükséges, az egység fizikai értelemben is megnyitásra kerül. Kazettás file-ok esetén ez a címke írását/olvasását jelenti. Lemezes file-ok esetén a paraméterek parancsként a buszra kerülnek.

Szintaxis: OPEN <aritmetikai kifejezés> [, <aritmetikai kifejezés>] [, <aritmetikai kifejezés>] [, <sztring kifejezés>] . Az első paraméter kötelező. Értéke az 1-255 intervallumba kell, hogy essen. Ez a paraméter a logikai file-szám. A második kifejezés az egység számát adja, ez a 0-15 intervallumba eshet. A harmadik kifejezés a megnyitási mód, értéke ugyancsak a 0-15 intervallumba esik. A <sztring kifejezés> értéke parancsként a buszra kerül.

Példák:

```
100 OPEN 10 : REM = OPEN 10,1,0
101 :
```



```

110 OPEN 15,8,15: REM A HIBACSATORNA MEGNYITASA
120 OPEN 1,8,4,"$": REM A # PSZEUDO-FILE MEGNYITASA
130 OPEN 2,8,2,"FILE,S,W": REM A FILE MEGNYITASA IRASRA
200 INPUT "HOVA KERI AZ OUTPUTOT";D
210 OPEN D,D
220 .... AZ EREDMENYEK KIIRASA ....
230 CLOSE D

```

READY.

Az első példa egy kazettás file-t nyit meg. /Mivel a kazettás file megnyitása a file fejlécének (header) olvasásával vagy írásával jár együtt, ezért kazettán egyszerre csak egy file lehet nyitva./ A fenti utasítás hatására a fejléc a kazetta pufferba kerül és ott /pl. PEEK-kel/ megvizsgálható. A második példában a 8-as lemezegységen négy file-t /15,1,2,3 logikai file-számokkal/ nyitunk meg. A harmadik példában, D értékétől függően, vagy a képernyőt, vagy a nyomtatót nyitjuk meg. Csak olyan szerkesztő karaktereket használhatunk, amelyek mindkét egységen ugyanugy hatnak.

A C-64 a következő megnyitási módokat értelmezi:

Egység	Egységszám	Megnyitási mód	Parancs
Billentyűzet	0		
Kazettás magnó	1	0 = input 1 = output 2 = output + EOT	A file neve
Modem	2	0	Kontrol regiszterek
Képernyő	3	0,1	
Nyomtató	4/5	0 = grafikus jelek 1 = normál	Kiirandó szöveg
Lemezegység	8-11	2-14 = adatcsatorna 15 = parancs csatorna 0 = SAVE 1 = LOAD	Meghajtó , file neve, file típusa, írás/olvasás

Mint a fenti táblázat is mutatja, számos esetben ezek a paraméterek tetszőlegesen megválaszthatók. Lemezegység esetén a 3-14 csatornák tetszőlegesen használhatók.

Az OPEN utasítás hatására az első három paraméter értéke beke-
rül a nyitott file-okat tartalmazó táblázatokba. Ezek a követ-
kezők:

1. Logikai file-számok /\$259-\$262, 601-610/.
2. Egységyszámok /\$263-\$26C, 611-620/
3. Megnyitási módok /\$26D-\$276, 621-630/

/\$98/ tartalmazza a nyitott file-ok számát. Az OPEN utasítás kiadásakor először ellenőrzi az interpreter, hogy a táblázatban szerepel-e ilyen sorszámú logikai file. Ha igen ?FILE OPEN hibaüzenettel térünk vissza a szövegszerkesztőbe. Ha mind a 10 logikai file nyitott, akkor ?TOO MANY FILES hibaüzenetet kapunk. Az OPEN utasítás még a következő címeket használja:

/\$B8/ = logikai file-szám
/\$B9/ = megnyitási mód
/\$BA/ = egységyszám
/\$BB/ = a parancs sztring kezdetére mutat
/\$B7/ = a parancs sztring hossza

Végrehajtás: A paraméter-értékek kiszámítása és a fenti memó-
riarészekbe való tárolása után az ST értékét nullára állítja,
majd a fenti értékeket beírja az OPEN-táblákba. Ezután kerül
sor a parancs sztring feldolgozására és az ennek megfelelő
tevékenység végrehajtására.

Hibalehetőségek: A paraméterek kiértékelése közben különféle
hibajelzést kaphatunk. Programhiba után célszerű az összes
file-t lezárni, különben újraindítás után ?FILE OPEN ERROR hiba-
jelzést kaphatunk. Nem megfelelő működést eredményezhet a meg-
nyitási mód, illetve a parancs sztring szerkezetének pontatlan
ismerete.

ORRövidítés: NincsToken: \$B0 (176)Belépési pont: \$AFE5(45029)Mód: Mind parancs, mind program módban használható.

Két aritmetikai kifejezés logikai 'vagy'-át határozza meg; előbb azonban a kifejezések értékét 2 byte-os egész számmá konvertálja. A 'vagy' művelet ezután bitenként, egymástól függetlenül hajtódik végre. Ha mindkét kifejezés logikai volt, akkor az eredmény 0 /hamis/ vagy -1 /igaz, \$FFFF/ lesz.

Szintaxis: <aritmetikai vagy logikai kifejezés >OR <aritmetikai vagy logikai kifejezés>.

Példák:

```
100 IF A<0 OR A>2000 THEN A$="ROSSZ"
200 PRINT -1 OR 1234
210 PRINT 380 OR 75
300 PRINT 380 OR 75
400 A%=12+(A<0 OR A>7)*(X-9)
```

READY.

Az első és negyedik sorban az OR tipikus használatát látjuk. Az első egy egyszerű feltételes értékadás. A negyedik sorban a feltételes értékadás az aritmetikai kifejezésbe épül be, felhasználva, hogy a logikai kifejezések egyben aritmetikai kifejezések is. A második és harmadik sor az OR aritmetikai kifejezésekkel való használatát mutatja be. A második sor eredménye -1, hiszen -1 = \$FFFF alakban kerül tárolásra, és itt mindegyik bit magas. Az OR művelet eredménye így \$FFFF=-1 lesz. A harmadik példa a 00000001 01111111 bit-képet adja, ami 383.

Az 'OR' művelet a műveletek közt a legalacsonyabb rendű, így utoljára hajtódik végre.

Végrehajtás: A TEST byte értéke \$FF lesz, majd az AND utasításban leírtak hajtódnak végre.

Hibalehetőségek: Megegyeznek az AND utasításéival.

PEEK

Rövidítés: pE

Token: \$C2 (194)

Belépési pont: \$B80D (47117)

Mód: Mind parancs, mind program módban használható.

A PEEK egyváltozós aritmetikai függvény, amelyik az argumentumaként megadott sorszámú címen levő byte értékét adja.

Szintaxis: PEEK(<aritmetikai kifejezés>). A kifejezés értékének a 0-65535 intervallumba kell esnie.

Példák:

```
PRINT " "CHR$(34);:FOR J=2048 TO 3047:PRINTCHR$(PEEK(J));:NEXT
FOR J=0 TO 999 : POKE 1024+J,PEEK(2048+J):NEXT
PRINT PEEK(SID+24)
POKE VIC+23,PEEK(VIC+23) OR 16
```

Az első két példa a BASIC szöveg első 1000 byte-ját írja ki a képernyőre, abban a formában, ahogy a memóriában tárolja az interpreter. A különbség nemcsak a használat módjában van /az egyik parancs, a másik program módban szerepel/, hanem a kiírás módjában is: az első 'nyomtat', a másik 'POKE'-ol. Az első megoldás így furcsa hatásokat idézhet elő. Például ha egy 34 kódu CLR kerül kiírásra a képernyő törlődik.

Az utolsó két példa azt mutatja, hogyan lehet a PEEK utasítást a perifériális egységek kezelésére használni. A példák a 7. és 8. fejezet mintapéldáiból valók és a hanggenerátor és a video-chip bizonyos regisztereit olvassák.

'Dupla-pontosságú' PEEK-re gyakran van szükség. Ezt a

$$\text{DEF FN DEEK}(X) = \text{PEEK}(X) + 256 * \text{PEEK}(X+1)$$

függvénnyel definiálhatjuk.

Végrehajtás: A rutin a (\$14) mutató tartalmát elmenti a verembe, majd kiértékeli az argumentumot, 2 byte-os egész számmá alakítja és ezt tölti a (\$14) mutatóba. A megfelelő byte /LDA \$14X/ az akkumulátorba kerül. Végül ezt egy rutin lebegőpontos alakra konvertálja.

Hibalehetőségek: Az aritmetikai kifejezések a 0-65535 intervallumba kell esnie. Ha ez nem teljesül, hibajelzést kapunk. Vannak olyan címek, amelyek olvasása nullázza a regisztert. Ilyenkor célszerű egy X=PEEK MEM utasítást, majd az X értékét használni.

POKE

Rövidítés: p0

Token: \$97 (151)

Belépési pont: \$B823 (47139)

Mód: Mind parancs, mind program módban használható.

Az utasítás segítségével egy tetszőleges RAM címre közvetlenül beírhatunk egy 0-255 közti egész számot.

Szintaxis: POKE <aritmetikai kifejezés>, <aritmetikai kifejezés>. A két kifejezés a memória címét és a tárolandó byte-ot jelenti. Ennek megfelelően az első értékének a 0-65535, a második értékének pedig a 0-255 intervallumba kell esnie.

Példák:

```
10 FOR J=1 TO 255: POKE 1023+J,J: POKE 55295+J,14 :NEXT
30 DATA 162,0,138,157,0,4,232,208,249,96
40 FOR J=828 TO 837:READ X: POKE J,X:NEXT
```

```

500 INPUT Y:FOR X=0 TO 9: POKE 2067,X+Y: NEXT
600 FOR J=200 TO 9E9: POKE J,170
610 IF PEEK(J)=170 THEN NEXT
700 POKE SID+24,15

```

READY.

Az öt példa messze nem érzékelteti azokat a lehetőségeket, amit a POKE utasítás biztosít. A 6-8. fejezetek példái ezt ékesen bizonyítják: a legtöbb perifériális eszköz csak POKE segítségével programozható.

Az első példa a képernyő tetejére kiírja mind a 256 lehetséges karaktert. A második példa azt mutatja, hogyan lehet a DATA utasításban tárolt gépi kódú programot 'betölteni' a POKE utasítás segítségével /A SYS 828 hatására ugyanaz történik, mint ami az első példában./ A harmadik példában egy önmódosító BASIC rutin részét látjuk, amelyik 10 byte-ot átmásol a memóriában. A következő példa a memória megadott részét ellenőrzi. Amikor a visszaolvasott érték már nem 170, az vagy a memória végét, vagy hibás címet jelent. Utolsó példánk a hanggenerátor hangerejét 15-re /a maximumra/ állítja.

Egy 'dupla-pontosságú' POKE függvényként nem írható fel. Egy egyszerű alprogram, amelyik ezt elvégzi, a következő:

```
POKE Z1,Z2 - INT(Z2/256) * 256: POKE Z1+1, Z2/256
```

Végrehajtás: A paramétereket a WAIT utasítással közös rutin értékeli ki, amelyik az első paramétert 2 byte-os egész számmá konvertálja, majd ezt a mutatót betölti /\$14/-be. Ezután kerül sor a vessző /,/ majd a második paraméter ellenőrzésére. Amennyiben a paraméter a 0-255 intervallumba esik, a megfelelő byte az X regiszterbe töltődik. Ezután kerül sor az indirekt értékadás végrehajtására.

POSRövidítés: NincsToken: \$B9 (185)Belépési pont: \$B39E (45982)Mód: Mind parancs, mind program módban használható.

Kiszámítja a kurzor helyzetét az aktuális képernyő sorban. A lehetséges érték 0-255. Ez nem a képernyő 40 karakteres sorában levő pozíció, hanem annak a mértéke, amennyivel a kurzort a sor elejétől elmozgattuk. Programsorok esetén ez maximum 30 lehet.

Szintaxis: POS (<kifejezés>). A POS-nak ál-argumentuma van, amelynek értéke, típusa lényegtelen. Általában POS(\emptyset)-t használunk.

Példák:

```
10 IF POS(0)>30 THEN PRINT CHR$(13)
20 PRINT TAB(10)POS(0); PRINT SPC(10)POS(0)
30 PRINT LEFT$("          ",12-POS(1));K$
```

READY.

POS talán a leghasznavehetetlenebb BASIC alapszó. Példánkban ellenőrizzük, hogy nem vagyunk-e a sor végén, s ha igen, egy [RETURN]-t írunk ki. A POS ugyanazokat a paramétereket használja, mint a TAB(. Így a POS utasítás nyomtatókkal kapcsolatban csak egy és ugyanazon fizikai soron belül használható.

Végrehajtás: A rutin a \emptyset . lapról az Y regiszterbe tölti a kurzor adott sorban levő helyzetét mutató byte-ot /\$D3, 211/. Ezt követően egy ROM rutin a byte-ot lebegőpontos számmá alakítja és a #1 lebegőpontos akkumulátorba teszi.

Hibalehetőség: Nincs.

PRINTRövidítés: ?Token: \$99 (153)Belépési pont: \$AA9F (43679)Mód: Mind parancs, mind program módban használható.

Kiszámítja a paraméterek értékét és az utasításban megjelölt formában kinyomtatja az elsődleges outputra /általában a képernyőre/.

Szintaxis: PRINT <nyomtatási kép>. A <nyomtatási kép> aritmetikai és/vagy sztring kifejezések sorozata, amelyeket egymástól a következő szeparátorokkal választunk el:

SPC(<aritmetikai kifejezés>);

TAB(<aritmetikai kifejezés>);

szóköz;

vessző /, /;

pontosvessző /; /;

ahol ez megengedett, nem kell szeparátor.

Példák:

```
10 FOR J=0 TO 255: PRINT CHR$(J):NEXT
20 FOR J=0 TO 100: PRINT J: NEXT
30 PRINT X+Y;124;P*G*(1-V)
40 PRINT "HELLO";A$B$U$U1$;MID$(X$,2)
50 PRINT TI;TI$;ST
```

A fenti 5 sor a PRINT utasítás szinte valamennyi lehetőségét bemutatja, kivéve a SPC és TAB használatát /ezek a megfelelő kucsszónál találhatók/. Az első példa 256 karaktert ír ki a képernyőre. Mivel ezek közt vezérlő karakterek is lehetnek, a kiírás közben a képernyő törlődhet, színes karakterek íródhatnak ki stb. A 2. utasítás a vessző /,/ használatát mutatja be. A következő mennyiség kiírása a legelső tabulálási

ponttól kezdődik. Ha a kinyomtatandó mennyiségeket pontosvesszővel /;/ választjuk el egymástól, akkor a nyomtatás a következő karakterhelytől kezdődik. A 40 karakter hosszú sorban 4 tabulálási pont van: 0, 10, 20, 30. Ennek megfelelően a 2. példa a 100 számot 25 sorban írja ki. A 3. és 4. példa az aritmetikai és sztring kifejezések használatát mutatja be. Az aritmetikai kifejezések értéke a STR\$(X)-nek megfelelő alakban kerül kiírásra /az utolsó karakter egy [CRSR⇒]./ Ilyen módon még ha pontosvesszővel /;/ elválasztott számokat is nyomtatunk ki, egy szóköz kerül közéjük. A 4. példa a szeparátorok elhagyását is bemutatja. A \$, % és a tömbváltozók végét jelző záró zárójelet az interpreter terminátornak tekinti, ezért ezek után a pontosvessző elhagyható. Utolsó példánk azt mutatja be, hogy a PRINT utasítás felismeri a fenntartott változókat /TI, TI\$, ST/ és a pi-t.

A szeparátorok elhagyása több esetben is lehetséges, de ezeknek a használatát nem javasoljuk:

```
10 PRINT (2)(7)      :REM 2 7
20 PRINT 1.2..3      :REM 1.2 0 .3
30 PRINT KESZ.       :REM 0 0
40 PRINT ;,L*K;4     :REM          0 4
```

READY.

Vezérlő karakterek használata Mint már a 2. fejezetben említettük, a PRINT utasításban szereplő sztring kifejezések vezérlő karaktereket is tartalmazhatnak, ezek 'kinyomtatása' a megfelelő vezérlő funkció végrehajtásával jár. Ugyanezt a hatást a CHR\$(X) alakú sztringek használatával is elérhetjük. Például

```
PRINT CHR$(18) "HAHO" CHR$(146) "HAHO"
```

az első HAHO-t inverz alakban írja ki.

A képernyőn nem biztos, hogy megjelenik a PRINT utasítással kiírt szöveg, mert az, amit a képernyőn látunk, nemcsak a képernyő memória tartalmától függ. Részletesen a 7. fejezetben szólnunk a C-64 grafikus lehetőségeiről.

Végrehajtás: A következő ábrán nagyjából bemutatjuk, hogyan is dolgozik a PRINT utasítás. A rutin maga nem hosszú, de egy tucat további ROM rutint használ.

A PRINT utasítást kezelő programrészek együtt elég nagy terjedelmet foglalnak el a ROM-ban.

Hibalehetőségek: A SPC(és TAB(paraméterei egész részének 0-255 intervallumba kell esnie. Amennyiben az utasításban nincs szintaktikus hiba, végrehajtódik. Ez persze messze nem jelenti azt, hogy olyan képernyőt kapunk, amilyent megterveztünk.

PRINT

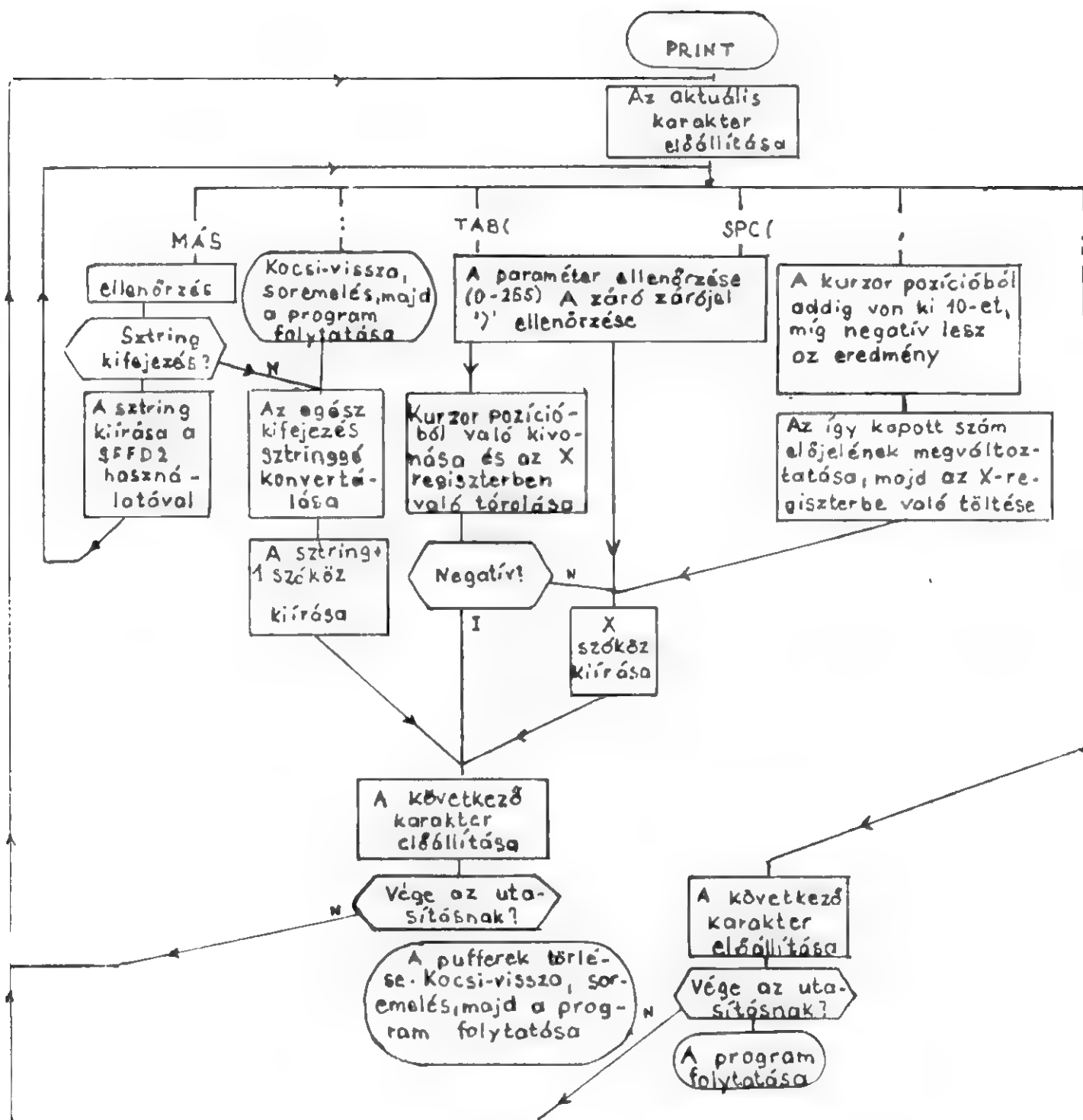
Rövidítés: pR /beleértve a #-t is/ Token: \$98 (152)

Belépési pont: \$AA7F (43647)

Mód: Mind parancs, mind program-módban használható.

Kiszámítja a paraméterek értékét és a megadott formában kiírja az utasításban specifikált logikai file-ba.

Szintaxis: PRINT # lf, <nyomtatási kép>. lf a logikai file szám. A <nyomtatási kép> alakja azonos a PRINT utasításban szereplővel.



A PRINT blokkdiagramja

Példák:

```

100 OPEN 4,4
110 PRINT#4,"ARAJANLAT"
200 OPEN 1,4,1:OPEN 2,4,7
210 PRINT#1,"ARAJANLAT";PRINT#2,"ARAJANLAT"
1000 OPEN 5,8,5,"ADATOK,S,W"
1010 FOR J=1 TO 10: READ A: PRINT#5,J;A: NEXT J
1020 CLOSE 5
1030 DATA 1,2,3,4,5,6,7,8,9,10

```

A fenti példák csak a nyomtató és a lemezegység használatát mutatják be, de hasonlóan lehet a többi perifériás egységet is programozni. Az első példa a nyomtatóra ír.

A második példában két csatornát nyitottunk meg a nyomtatón és felváltva használjuk a PRINT# utasítással.

Utolsó két példánk a lemezes file-ok használatát mutatja be.

Végrehajtás: A rutin összesen két gépi kódu utasításból áll. Az első meghívja a CMD-t végrehajtó rutint. Ez ellenőrzi a paramétereket és végzi a kiírást. A második meghívja az a rutint, amelyik 'UNLISTEN' parancsot küld az eszköznek, amelyre a kiírás történt.

A CMD hatására az eszköz 'LISTEN' állapotban marad.

Hibalehetőségek: A SPC ; a TAB és a vezérlő karakterek a kiírás során nehézségeket okozhatnak és a file tartalma lehet, hogy nem egyezik meg a kívánttal.

READ

Rövidítés: RE

Token: \$87(135)

Belépési pont: \$AC05(44037)

Mód: Mind parancs, mind program módban használható.

A DATA utasításokban definiált értékeket olvassa és rendel hozzá az utasításban megadott változókhoz.

Szintaxis: READ <változólista> . A <változólista> tetszőleges típusu, egymástól vesszővel elválasztott egyszerű- vagy tömbváltozókat tartalmaz. Legalább egy változó megadása kötelező.

Példák:

```
10 DATA 10 : READ N
20 FOR I=1 TO N: READ X1%(N): NEXT I
110 READ CIM: IF CIM<0 THEN LO%=(CIM+U)AND 255: HI%=(CIM+U)/256
200 FOR I=1 TO 10 READ A: PRINT A,: NEXT
```

Az első példa azt illusztrálja, hogyan lehet a DATA utasítások első elemeként megadni az összes tárolt érték számát. A második példa egy áthelyezhető gépi kódu program töltését mutatja. Amennyiben az érték negatív, a címet U% hozzáadásával kell előállítani. Utolsó példánk a READ parancs módban való használatát mutatja; a parancs a tárolt program következő 10 adatát olvassa és írja ki a képernyőre.

Végrehajtás: A rutinnak közös részei vannak a GET, GET/ , INPUT és INPUT/ rutinokkal. A Ø. lapon található jelzőbyte értékét \$98-ra állítja be a READ. /Részleteket lásd a GET leírásánál/.

Hibalehetőségek: Ha a változó és a DATA utasításban talált érték típusa nem felel meg egymásnak, akkor ?TYPE MISMATCH vagy ?SYNTAX ERROR hibajelzést kapunk. Ha a DATA-ban levő adatok elfogytak ?OUT OF DATA hibajelzést kapunk.

REM

Rövidítés: nincs

Token: \$8F(143)

Belépési pont: \$A93A (43322)

Mód: Mind parancs, mind program-módban használható.

Lehetővé teszi megjegyzések beírását a program szövegébe. Az interpreter a REM észlelésétől a sor maradék részét már nem hajtja végre, hanem a következő sor elejéről folytatja a program végrehajtását.

Szintaxis: REM <karaktersorozat> . A <karaktersorozat> tetszőleges karaktereket tartalmazhat; beleértve a kettőspontot is /:/. A REM után - a következő sorig-bármilyen állhat.

Példák:

```
7000 REM *** FOPROGRAM ***
7010 GOSUB 1000: REM *** KEZDOERTEKEK
7020 GOSUB 2000: REM *** SZAMITASOK
7030 GOSUB 3000: END: REM *** EREDMEYEK
7040 REM *** SZUBROUTINOK
```

```
10 REM
*** FOPROGRAM ***
```

```
X$=A$+CHR$(13)+A$+B$:GOSUB 2234:REMUB 2357
```

Az első példáink a REM leggyakoribb felhasználását mutatják, amikor az utasítások hatását írja le. A 10 sorban a REM-mel kapcsolatos egyik legegyszerűbb trükköt mutatjuk be; a REM után egy [RETURN] karaktert POKE-oltunk be, aminek hatására a listázás kor a megjegyzés új sorban jelenik már meg.

Az utolsó sor a REM parancs módban való használatát mutatja be, egy program sorát kilistáztuk, töröltük a sorszámot és a nem kívánt rész elé egy REM-et helyeztünk el. A [RETURN] megnyomása után a parancs a REM-ig hajtódik végre.

A LIST parancs a REM sorokban talált tokeneket nem grafikus jelként, hanem eredeti alapszavának megfelelően listázza ki.

Végrehajtás: A rutin megkeresi a program végét jelző 0 byte-ot, majd a CHRGET mutatóját ennek megfelelően átállítja. A rutin az IF-et végrehajtó programrész közepén található, hiszen ha az IF-ben szereplő logikai kifejezés hamis a sor maradék részét ugyancsak át kell lépni.

Hibalehetőség: Ha a REM-sor [SHIFT L] karaktert tartalmaz, a listázás ?SYNTAX ERROR hibaüzenettel megszakad. /ROM-hiba/.

RESTORE

Rövidítés: reS

Token: \$8C (140)

Belépési pont: \$A81C (43303)

Mód: Mind program, mind parancs módban használható.

Az utasítás hatására a READ az első DATA sorból olvassa a következő értéket.

Szintaxis: RESTORE.

Példák:

```
10 DATA GEPI KOD1,169,0,141,....
1340 RESTORE: FOR I=1 TO 9E9: READ X$
1345 IF X$<>"GEPI KOD1" THEN NEXT
1350 FOR L=828 TO 912:READ X:POKE L,X:NEXT
2010 READ X$: IF X$="END" THEN RESTORE
```

Az első példa azt mutatja, hogyan lehet a DATA utasítások közt egy adott részt megtalálni. Először végrehajtjuk a RESTORE utasítást, majd addig olvassuk a DATA-kat, míg a "GEPI KOD1" sztringet meg nem találjuk. Ezt követik a gépi kódú rutin byte-jai.

A második példában az utolsó DATA tétel egy "END" sztringet tartalmaz. Ennek az olvasása után a RESTORE a DATA mutatót az első utasításra állítja.

A NEW, RUN, CLR utasítások egyben egy RESTORE-t is végrehajtanak. Parancs módban a RESTORE-t így csak a GOTO <sorszám> utasítás előtt érdemes használni.

Végrehajtás: A 'BASIC program eleje' /\$2B-2C/ mutatót eggyel csökkenti és betölti a \$41-42 címekre. Így ez a mutató egy 0 byte-ra mutat.

Hibalehetőség: nincs

RETURNRövidítés: reTToken: \$8E(142)Belépési pont: \$A8D1(43217)Mód: Mind parancs, mind program-módban használható.

Az utasítás hatására a program vezérlése az utolsó GOSUB utasítást követő első utasításra kerül.

Szintaxis: RETURNPéldák:

```
10 INPUT X : GOSUB 1500 : GOTO 10
1500 X=XINT(( X-.005 / RQ ) +1) * RQ
1510 PRINT X ; : RETURN
```

Egyetlen példánk az 1500. sorban kezdődő kerekítési eljárás ellenőrzését mutatja be; a 10. sorban különböző értékeket adhatunk be a programnak és kerekített alakját ellenőrizhetjük.

Végrehajtás: Az utasítás szintaxisának ellenőrzése után a rutin végigolvassa a vermet FOR és GOSUB tokeneket keresve. A FOR tokeneket és a hozzájuk tartozó információt törli a veremből. A GOSUB megtalálása után visszatölti a megfelelő értékeket, majd megkeresi a következő kettőspont `:/` vagy `Ø` byte karaktert onnan folytatja a program futását. Ha például az `ON L GOSUB 10,20,30 : PRINT X` utasítás segítségével hajtottunk végre egy szubrutinhívást, akkor a veremből való visszatöltés után a CHRRET mutatója valahová a számok közé mutat. Ezt a rutin a kettőspontig növeli.

Hibalehetőségek: Ha az eljárás nem találja meg a visszatérési címet, `?RETURN WITHOUT GOSUB` hibajelzést kapunk. A RETURN végrehajtása törli a verem tetején levő FOR-ra vonatkozó informá-

ciókat. A

```
10 GOSUB 1000 : NEXT J
1000 FOR J = 0 TO 10 : RETURN
```

program végrehajtása közben ezért ?NEXT WITHOUT FOR hibajelzést kapunk.

RIGHT\$

Rövidítés: rI /beleértve a \$-t/ Token: \$C9(201)

Belépési pont: \$B72C (46892)

Mód: Mind parancs, mind program-módban használható.

A sztring függvény egy adott sztring néhány utolsó karakteréből egy új sztringet képez.

Szintaxis: RIGHT\$(<sztring kifejezés>, <aritmetikai kifejezés>) .

Az aritmetikai kifejezés nem lehet 255-nél nagyobb. A függvény az első paraméter kiértékelése után kapott sztring - a második paraméterben megadott szám - utolsó karakteréből képez egy új sztringet. Ha a paraméterként megadott sztringben nincs annyi karakter, akkor valamennyi karaktert felhasználja.

```
Legyen          X$ = "KOVACS PETER"
                Pozíció = 123456789012

Ekkor          RIGHT$(X$,5) = "PETER"
                RIGHT$(X$,25) = X$.
```

Példák:

```
10 PRINT RIGHT$("ABRAKADABRA",3) :REM = BRA
20 PRINT RIGHT$("          "+STR$(N),10)
```

READY.

Az első példa a RIGHT\$ hatását mutatja be, a második az N értékének megfelelő számot jobbra tömörítve 10 karakterpozíción írja ki.

A RIGHT\$ függvény 'felesleges' lévén

RIGHT\$ ~~(X\$,N)~~ = MID\$(X\$, LEN(X\$) - N + 1)
(X\$,N)

Végrehajtás: A sztringkifejezés kiértékeléseként megkapjuk a sztring paramétereit /hosszát, kezdetét/. Ennek segítségével a rutin kiszámítja az eredménystring hosszát, majd az X\$-ből elhagyandó karakterek számát. Ezt betölti az A regiszterbe, majd a LEFT\$ rutint hajtja végre.

Hibalehetőségek: A paraméterek értéke nem eshet a megadott értékhatárokon kívül.

RND

Rövidítés: nincs

Token: \$BB (187)

Belépési pont: \$E097 (57495)

Mód: Mind parancs, mind program-módban használható.

A 0-1 zárt intervallumba eső pseudo-véletlen számokat generál.

Szintaxis: RND(<aritmetikai kifejezés>). Az aritmetikai kifejezés értékének csak az előjele számít.

Példák:

```
10 FOR J=0 TO 3000*RND(1):NEXT
100 FOR J=1 TO 100*RND(1): READ X$: NEXT
200 X=(B-A)*RND(1)+A
```

READY.

Az első példa a program futását maximum három másodpercig - de véletlen hosszúságu időtartamra - felfüggeszti. A második példa egy 100 elemű lista véletlenszerűen kiválasztott elemét olvassa. Utolsó példánk az [A,B] intervallumba eső véletlen számokat állít elő.

Az RND utasítás természetesen nem állít elő 'igazi' véletlen számokat. Az argumentum előjele határozza meg, milyen eljárás segítségével számítja ki az interpreter a következő számot.

Végrehajtás: Az utoljára előállított véletlen számot az interpreter külön tárolja, és bizonyos számelméleti függvények segítségével kapja meg a következő véletlen számot. Amennyiben a RND függvény argumentuma pozitív, az interpreter a következő véletlen számot generálja. Negatív argumentum az első véletlen számot állítja elő; `X=RND(-1) : PRINT X` értéke mindig ugyanaz. Végül `RND(0)` a CIA chipek óra-regisztereit használja fel véletlen számok előállítására. Ez csak látszólag véletlen, hiszen ha egy cikluson belül ismételten használjuk a `RND(0)` utasítást, akkor bizonyos szabályosság azonnal jelentkezik a végrehajtási időben.

Hibalehetőségek: Az aritmetikai kifejezés kiértékelése okozhat csak hibát.

RUN

Rövidítés: rU

Token: \$8A(138)

Belépési pont: \$A870 (43120)

Mód: Mind parancs, mind program-módban használható.

A memóriában tárolt programot az elejéről, vagy egy adott ponttól kezdve végrehajtja. A változók előző értéke elvész.

Szintaxis: RUN [`<sorszám>`]

Példák: RUN
 RUN 1000
 5000 IF X <> 0 THEN RUN

Az első két sor az utasítás parancs módban való használatát szemlélteti. Az utolsó példában, ha $X \neq 0$, akkor a RUN utasítás törli az összes változót és a program előről kezd el futni. Ha a RUN utasítást nem követi egy kettőspont `/:/` vagy egy \emptyset byte, akkor az interpreter sorszámot tételez fel. Így RUN X és RUN "PRG" hatása egyaránt RUN \emptyset /lásd a GOTO utasítást !/.

A `[SHIFT-RUN]` billentyű lenyomása ekvivalens a `LOAD[RETURN]` RUN `[RETURN]` billentyűzésével, tehát a szalagról való betöltés után automatikusan futtatja a programot.

Végrehajtás: A rutin a CHRGET mutatóját eggyel a 'BASIC-program kezdete'elé állítja, törli az összes változót, a megnyitott file-okat, a program elejére állítja a DATA mutatót, ki-nullázza a vermet. Ezután belép a BASIC utasítások végrehajtását ellenőrző ciklusba. /lásd a 3. Fejezetben/. Sorszám megadása esetén a törlések végrehajtása után a GOTO rutinra kerül a vezérlés.

Hibalehetőség: Ha a RUN utasításban megadott sorszámú program-sor nincs ?UNDEF'D STATEMENT ERROR hibajelzést kapunk.

SAVE

Rövidítés: SA

Token: \$94 (148)

Belépési pont: \$E155 (57685)

Mód: Mind parancs, mind program módban használható.

Az utasításban megadott nevű file-ba elmenti a memóriában tárolt programot.

Szintaxis: Megegyezik a LOAD szintaxisával. A második <aritmetikai kifejezés> egyedül a kazettás egység esetén érdekes, ha értéke 2, akkor a file elmentése után még egy E-O-T jel is kiíródik a szalagra. /Lásd a 6. Fejezetet/

Példák:

```

SAVE      REM NEV NELKUL A KAZATTAN
SAVE "",8:REM HIBAJELZEST KAPUNK, "" NEM NEV
SAVE "PROG=5",8:REM A PROGRAM NEVE PROG=5 LESZ A LEMEZEN
SAVE "@:REGI",8:REM LEMEZEN MAR MEGLEVO FILEBA
12 SAVE "TEST"+TI$,8

```

A fenti példák önmagukért beszélnek. Az utolsó példát emelnénk csak ki; a programrész egy olyan rutin része, amelyik időről-időre a tárolt programot kiírja a memóriából. Hogy ezek a file-ok különbözőek legyenek a név utolsó hat karakterét a TI\$ adja, ami mindig más és más.

Végrehajtás: A rutin ST-t nullázza, majd előállítja az utasítás paramétereit. Ezt követően áttölti a 'BASIC program eleje', 'BASIC változók eleje' mutatókat saját munkaterületére. A memória e két érték közé eső része kerül elmentésre. Ezt követi az egységszám ellenőrzése. Ha ez 0 vagy 3 /billentyűzet, képernyő/ akkor ?DEVICE NOT PRESENT hibajelzést kapunk. Az 1, 2. és 8-15. egységszámoknak /kazettás egység, modem, lemezegységek stb./ megfelelő három további rutin hajtja végre a valódi mentést. /Programok a nyomtatón is menthetők, de meglehetősen furcsa nyomtatási képet kapunk./

Kazettás egység esetén /egységszám = 1/ az interpreter a képernyő-re kiírja a

PRESS PLAY AND RECORD

üzenetet és a megfelelő kezelőgombok lenyomása után kezdi meg a mentést.

Hibalehetőségek: Ha az utasításban megjelölt egység nincs bekapcsolva ?DEVICE NOT PRESENT hibaüzenetet kapunk. A kazettás egységre való kimentés aszinkron. Ha például csak a PLAY billentyűt nyomjuk meg az utasítás hibajelzés nélkül végrehajtódik,

bár a szalagra a program természetesen nem kerül fel. Ha a lemezen már meglevő file-ba mentjük a programot, akkor csak a " \emptyset : <név> " alakot használhatjuk. Ellenkező esetben ?FILE EXISTS hibajelzést kapunk.

SGN

Rövidítés: sG

Token: \$B4(180)

Belépési pont: \$BC39 (48185)

Mód: Mind parancs, mind program-módban használható.

Az aritmetikai függvény az argumentum előjelét számítja ki.

$$\text{SGN}(X) = \begin{cases} +1 & \text{ha } X > \emptyset, \\ \emptyset & \text{ha } X = \emptyset, \\ -1 & \text{ha } X < \emptyset. \end{cases}$$

Szintaxis: SGN (<aritmetikai kifejezés>).

Példák:

```
10 IF SGN(X)>0 THEN PRINT X;"POZITIV"
20 IF ABS(SGN(X))<>0 THEN PRINT X;"NEM NULLA"
30 ON SGN(X)+2 GOSUB 100,110,120
```

Az első két sor az SGN egyszerű használatát mutatja. A harmadik példa a FORTRAN aritmetikai GOTO utasítását szimulálja. /FORTRAN ekvivalens: IF(X) 100,200,300/.

Végrehajtás: Először egy rövid rutin az A regiszterbe 0-t, 1-t, vagy \$FF-et tölt az #1 lebegőpontos akkumulátor előjelétől függően. Ezt egy konvertáló rutin követ, amelyik a kívánt értéket végül is előállítja.

SIN

aritmetikai függvény

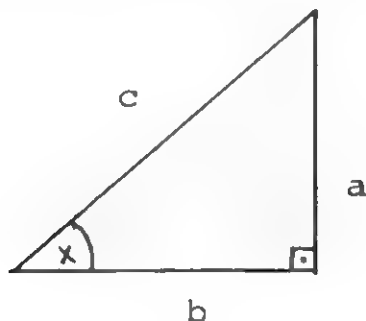
Rövidítés: SI

Token: \$BF(191)

Belépési pont: \$E263 (57963)

Mód: Mind parancs, mind program módban használható.

A radiánokban megadott szög szinuszát számítja ki. Az alábbi ábra illusztrálja x és $\sin(x)$ viszonyát:



$$\sin(x) = \frac{a}{c}$$

Szintaxis: SIN(<aritmetikai kifejezés>). A kifejezésnek szintaktikusan helyesnek kell lennie.

Példák:

```
10 PRINT SIN(1)      :REM =.841470985
20 PRINTSIN(360*PI/180): REM =0
100 PRINT "I":FOR I=1 TO 25:PRINT TAB(SIN(I/3)*19+20);"I":NEXT I
200 X=A+SIN(A):Y=A+SIN(A/2)*2
```

Az első két példa önmagáért beszél. A harmadik program egy szinuszgörbét rajzol a képernyőre. A negyedik példa egy rajzoló program része.

Az argumentum nagysága a végeredményt nem befolyásolja; a számítás során az interpreter elosztja ugyanis $2 * \pi$ -vel és a maradékkal számol tovább.

Végrehajtás: Az argumentum értékét $\text{mod } 2 * \pi$ az #1 lebegőpontos akkumulátorba helyezi a rutin, majd az argumentum előjelét a verembe tölti. A valódi számítást egy öt tagu hatványsor végzi; a végeredmény előjelét a végén állítja be a rutin.

Hibalehetőségek: Az aritmetikai kifejezés kiértékelése közben számos hibalehetőség adódik. Ha ez hiba nélkül befejeződött, a SIN rutin már hibátlanul lefut.

SPC(

Rövidítés: SP /beleértve a kezdő zárójelet/ Token: \$A6(166)

Mód: Mind parancs, mind program-módban használható

Adott számú szóközt, vagy [CRSR⇒] karaktert nyomtat.

Szintaxis: SPC(<aritmetikai kifejezés>). Az utasítás csak a PRINT utasításban szerepelhet. Az SPC és a (jel közt nem lehet szóköz. Az <aritmetikai kifejezés> értékének - lefelé kerekítés után - a 0-255 intervallumba kell esnie.

```
10 PRINT "□":FOR J=0 TO 20:PRINT "*"SPC(38)*":NEXT
20 FOR J=1 TO 19: PRINT SPC(J)*"SPC(38-J*2)*":NEXT
```

A fenti példák azt illusztrálják, hogy egy ciklusban használt SPC(utasítás segítségével hogyan rajzolhatunk a képernyőre. Az első program egy keretet rajzol, a második pedig egy V alakú alakzatot.

Az, hogy a SPC(szóközt vagy [CRSR⇒] karaktert nyomtat, a programból állítható. Ha a 19/\$13/. címen 0 található akkor a SPC(a kurzort mozgatja, ha ettől eltérő az érték, akkor szóközöket ír ki, ahogy ezt az alábbi program illusztrálja:

```
10 INPUT X
20 POKE 19,X
30 PRINT "□":FOR J=0 TO 30: PRINT "X":NEXT
40 PRINT "***TAB(10)***SPC(10)***"
50 END
```

```
***          ***          *****(X=1)
*****XXXXXXXXXXXXXXXXXXXXXXXXXXXX*(X=0)
```


Végrehajtás: A végrehajtást a PRINT rutin végzi. Amikor egy SPC(. tokenet talál, kiértékeli a kifejezés értékét. Az eredményt 1 byte-os egész számmá konvertálja és az X regiszterbe tölti, amelyik a kiíratandó karakterek számát adja.

Hibalehetőség: Ha az aritmetikai kifejezés értéke nem esik a 0-255 intervallumba ?ILLEGAL QUANTITY ERROR hibajelzést kapunk.

SQR

Rövidítés: sQ

Token: \$BA(186)

Belépési pont: \$BF71 (49009)

Mód: Mind parancs, mind program módban használható.

Az argumentum négyzetgyökét számítja ki.

Szintaxis: SQR(<aritmetikai kifejezés>) .

Példák:

```
10 PRINT SQR(2) :REM = 1.4142...
20 PRINT SQR(9) T2 :REM = 9
1000 X1=(-B+SQR(B*B-4*A*C))/(2*A)
1010 X1=(-B-SQR(B*B-4*A*C))/(2*A)
2000 T=SQR(X*X+Y*Y+Z*Z)
```

READY.

A SQR függvényre - éppen úgy, mint az EXP-re valójában nincs szükség, hiszen $SQR(X) = X \uparrow .5$.

Az első két példa az SQR hatását mutatja be. A következő két sorban az A,B,C együtthatóju másodfoku egyenlet megoldásait számítottuk ki. Utolsó példánk az (X,Y,Z) pontnak az origótól vett távolságát adja.

Végrehajtás: Az argumentum a #2 lebegőpontos akkumulátorba kerül, majd a rutin az #1 lebegőpontos akkumulátorba tölti a 0.5 értéket. Ezután hatványfüggvényt kiszámító rutin megfelelő részére ugrik a vezérlés.

Hibalehetőségek: Ha az argumentum értéke negatív
?ILLEGAL QUANTITY hibajelzést kapunk.

ST

Fentartott BASIC változó.

ST a perifériális egységek mindenkori állapotát írja le. ST jelentése az egység típusától függően más és más. A következő táblázat ST lehetséges értékét és azok jelentéseit foglalja össze.

Bit	Numerikus érték	Kazettás egység	Soros kimenet I/O	Kazettás egység Verify+Load
0	1		idő túllépés írásnál	
1	2		idő túllépés olvasásnál	
2	4	rövid blokk		rövid blokk
3	8	hosszu blokk		hosszu blokk
4	16	felismerhetetlen olvasási hiba		hiba
5	32	ellenőrző összeg hibás		ellenőrző összeg hibás
6	64	file vége jel	EOI	
7	- 128	szalag vége jel	az egység nincs bekapcsolva	szalag vége jel

Az ST értékét a GET, INPUT és a PRINT, továbbá a GMD, GET#, INPUT# és a PRINT# utasítások nullázzák. Az ST-ben tárolt információt valamennyi I/O utasítás után ellenőrizni lehet.

ST nullától különböző értéke nem feltétlenül jelent hibát /pl. ST=64/. Az ST értékét az interpreter egy byte-os számként a 144 /\$90/ címen tárolja.

STOPRövidítés: sTToken: \$90 (144)Belépési pont: \$A82E (43054)Mód: Mind parancs, mind program módban használható.

Megállítja a program futását és kiírja, hogy hányadik sorban állt meg a program. A program futása a CONT utasítással folytatható.

Példák: A STOP utasítást elsősorban a programfejlesztés stádiumában használjuk, hiszen kész programok esetén érdektelen, hogyan is dolgozik a program. A STOP utasítás segítségével töréspontokat helyezhetünk el a programban:

```
10 GOSUB 2000: STOP: GOSUB 3000: STOP
1254 IF G$="" THEN STOP
```

READY.

A STOP végrehajtása után a program változóit a PRINT utasítás segítségével ellenőrizhetjük. Ezután a CONT még működik. Ha egy régi vagy új változónak adunk értéket a CONT általában még mindig használható.

Végrehajtás: A rutin majdnem azonos az END utasítás rutinjával, azzal a különbséggel, hogy a végén a "BREAK IN" sztring is kiírásra kerül, amit a megfelelő sorszám követ.

Hibalehetőségek: nincsSTR\$Rövidítés: stR /beleértve a \$-t is/Token: \$C4 (196)Belépési pont: \$B465 (46181)Mód: Mind parancs, mind program módban használható

Az argumentumként megadott aritmetikai kifejezés értékének sztring alakját adja.

Szintaxis: STR\$(<aritmetikai kifejezés>) . A STR és a \$ jel közt nem lehet szóköz. Az <aritmetikai kifejezés> értéke a PRINT formátumának megfelelő alakban áll elő. Így például STR\$(.005) nem ".005", hanem " 5E-03".

Példák:

```
10 PRINTSTR$(555)+".00"      :REM = 555.00
20 N$="0"+MID$(STR$(N),2): PRINT N$
```

READY.

Az utasítást elsősorban a számok alakjának szerkesztésére használjuk. Az első példa egyszerűen csak szemlélteti a STR\$ hatását. A második példában a számokat megelőző szóközt töröljük a sztringből. N\$ kiírása az 1-nél kisebb számokat, például .05-t 0.05 alakban írja ki. A harmadik példa azt illusztrálja, hogy STR\$ sztring-függvény is, és a + műveletben használható.

Végrehajtás: A rutin először az argumentum típusát ellenőrzi, majd a PRINT végrehajtásának egyik rutinja kerül meghívásra, amelyik a kinyomtatandó számok sztring alakját állítja elő. Végül a rutin a verembe tölti a sztring paramétereit.

Hibalehetőség: A számok normális írása és a STR\$(N) alak nem mindig egyezik meg. Ebből szemantikus hibák adódhatnak.

SYS

Rövidítés: sY

Token: \$9E(158)

Belépési pont: \$E129 (57641)

Mód: Mind parancs, mind program módban használható.

A vezérlést a SYS argumentumán kezdődő gépi-kódu rutinnak adja át. A RTS utasítás végrehajtása után a vezérlés a BASIC-be kerül vissza.

Szintaxis: SYS <aritmetikai kifejezés> .

Példák: Az 5. - 10. Fejezetekben számos példát adunk használatára. A SYS az egyik utasítás, amelyikkel gépi-kódu alprogramokat hívhatunk. Ezek lehetőségeiről bővebben a 9. Fejezetben szólunk.

Végrehajtás: A paraméter értékét a rutin 2 byte-os egész számmá konvertálja és (\$14)-be tölti. Ezután egy indirekt ugró utasítás következik: JMP (\$0014). A visszatérés után a vezérlés a BASIC feldolgozó ciklusának elejére kerül.

Hibalehetőségek: Az utasítás hatására a program futásának ellenőrzése kikerül a BASIC interpreter hatálya alól. Ha a gépi-kódu rutint nem jól irtuk meg, a teljes C-64 'lemerevedhet'. Ilyenkor nincs mást tenni, mint kikapcsolni a gépet. SYS-t tartalmazó program kipróbálása előtt feltétlenül mentsük el a programot!

TAB(

Rövidítés: tA /beleértve a kezdő zárójelet is/ Token: \$A3(163)

Mód: Mind parancs, mind program módban használható.

Amennyiben a kurzor pozíciója az adott sorban kevesebb, mint a TAB(paraméterében megadott érték, addig a pozícióig szóközöket, vagy [CRSR↔] karaktereket nyomtat.

Szintaxis: TAB(<aritmetikai kifejezés>). Az utasítás csak a PRINT utasításban szerepelhet. A TAB és (jel közt nem lehet szóköz.

Példák:

```
10 PRINTTAB(LEN(N$)/2);N$
20 PRINT"R";TAB(255)
```

READY.

Az első példa a képernyő közepére tabulálja az N\$-ban tárolt üzenetet. A következő példa azt illusztrálja, hogy a TAB(hatása több soron keresztül is érvényesülhet.

Végrehajtás: Az utasítást a PRINT rutin megfelelő része hajtja végre. Amikor egy TAB(tokenet talál, kiértékeli a kifejezés értékét, majd kivonja belőle a kurzor pozícióját jelző byte értékét. Ha ez pozitív, betölti az X regiszterbe és ennyi szóközt, vagy [CRSR⇒] karaktert ír ki.

Hibalehetőség: Ha az aritmetikai kifejezés értéke nem esik a 0-255 intervallumba ?ILLEGAL QUANTITY ERROR hibajelzést kapunk.

TAN

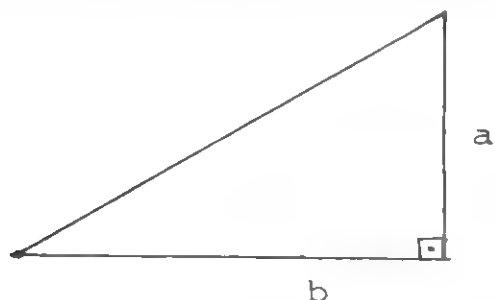
Rövidítés: nincs

Token: \$CO (192)

Belépési pont: \$E2B4 (58036)

Mód: Mind parancs, mind program-módban használható.

A radiánban megadott argumentumának tangensét számítja ki. Az alábbi ábra mutatja, hogyan lehet egy háromszög oldalaiból a $\text{tg}(x)$ függvényt kiszámítani.



$$\text{tg}(x) = \frac{a}{b}$$

Szintaxis: TAN(<aritmetikai kifejezés>)

Példák:

```
10 PRINT TAN(PI/2)      : REM OVERFLOW ERROR
20 PRINT TAN(45/57.29578): REM = TG(45)
30 X=(TAN(A)+TAN(B))/(1-TAN(A)*TAN(B))
```

READY.

Az első két példa a TAN hatását mutatja be, a másodikban a fokokban adott szöget először átszámítjuk radiánba. A harmadik példa a tangens függvényre vonatkozó addíciós tételt használja; X valójában $\tan(A+B)$.

Végrehajtás: Az argumentum értékének kiszámítása és ellenőrzése után a rutin a $\text{tg}(x) = \sin(x) / \cos(x)$ képlet felhasználásával számítja ki a függvény értékét.

Hibalehetőségek: Amikor $\cos(x) \approx 0$?OVERFLOW ERROR hibaüzenetet kaphatunk.

TI és TI\$

Fenntartott BASIC változók.

TI és TI\$ a C-64 belső órájának segítségével a bekapcsolás óta eltelt időt jelzik. TI valós változó, ami azonban csak olvasható. A 0. lapon található 'szabadon futó óra' másodpercenként körülbelül 60-szor kerül aktualizálásra, így TI/60 a gép bekapcsolása óta eltelt másodperceket mutatja. TI\$ egy hat karakteres sztring, amelynek két-két karaktere rendre az órát, a percet és a másodpercet adja. TI\$ értéke értékadó utasítással változtatható.

```
10 TI$="081500"
20 TI$=0$+P$+M$
100 T=TI
110 IF TI-TC120 THEN GOTO 110
1000 PRINT MID$(TI$,1,2);":";MID$(TI$,3,2);":";
1010 PRINT RIGHT$(TI$,2);PRINT" ";GOTO1000
5000 TI$="000000"
5010 REM <PROGRAM>
5020 PRINT TI/60 : END
```

Az első két példa azt mutatja, hogy TI\$ értékét a programból megadhatjuk. A sztringnek, ami TI\$ új értékét definiálja pontosan hat karakter hosszúnak kell lennie. A harmadik példa 2 másodpercnnyi időre felfüggeszti a program futását /ciklusban várakozik/. A negyedik példa az időt a digitális órákon szokásos xx:yy:zz alakban jelzi ki. Utolsó példa egy tetszőleges program futási idejének kiszámítására szolgál. A program futása előtt az órát TI\$="000000" értékadással kinullázza, majd a program végét jelző END előtt kiírja a futási időt másodpercekben.

Végrehajtás: A 'szabadon futó' óra a 0. lapon a 160-162. címen található és a hardver megszakító rutin aktualizálja. TI\$ értékét TI értékéből egy hosszadalmas ROM-rutin számítja ki. TI\$ értékét az interpreter a többi változó közt tárolja.

Hibalehetőségek: Ha olyan gépi-kódu rutint, vagy BASIC parancsot használunk, amelyik letiltja a megszakításokat, az 'óra' pontatlanná válik, késik.

USR

Rövidítés: us

Token: \$B7 (183)

Belépési pont: \$310 (784)

Mód: Mind parancs, mind program-módban használható.

Aritmetikai függvény, amelyik a felhasználó által definiált gépi-kódu programot hajtja végre.

Szintaxis: USR(<aritmetikai kifejezős>). Az aritmetikai kifejezés értékének a 0-65535 intervallumba kell esnie /lekerekítés után./ Ezenkívül a 784-786 címen egy gépi-kódu utasításnak kell lennie /ez általában JMP xxxx alaku/.

```
1. POKE 785,1+16+16 POKE 786,0
   GOTO USR 1
```


Szintaxis: VAL(< sztring kifejezés>). A <sztring kifejezés> értékeként egy legfeljebb 255 hosszú sztringet kell kapnunk.

Példák:

```

10 PRINT VAL("123.321")      : REM = 123.321
20 PRINT VAL("-123")          : REM = -123
30 PRINT VAL("1.2 E2")        : REM = 120
40 PRINT VAL("E")             : REM = 1
50 PRINT VAL("10000000000")    : REM = 1E+10
60 PRINT VAL(LEFT$(TI$,2))     : REM = 0-24 KOZT
70 PRINT VAL("123+321")        : REM = 150
80 PRINT VAL("1.2.3")          : REM = 1.2
90 PRINT VAL("")              : REM = 0
100 PRINT VAL("1")+VAL("2")    : REM = 3

```

READY.

Mint a fenti példák is mutatják, bármilyen is az argumentum értéke, a VAL függvény mindig előállít valamilyen számot. Ez gyakran lehet szemantikus hiba forrása. Utolsó példánk mutatja, hogy VAL aritmetikai függvény és ezért tetszőleges aritmetikai kifejezésben használható.

A VAL felismeri a +, -, E jeleket, a tizedespontot(.), és ezenkívül természetesen a 0-9 számjegyeket. Az első - szám részeként - nem értelmezhető karakter befejezi a sztring kiértékelését.

Végrehajtás: A rutin legnagyobb része egy sztring-szám konverter, amelyik végül is az #1 lebegőpontos akkumulátorban állít elő egy számot.

Hibalehetőség: nincs.

VERIFYRövidítés: vEToken:\$95 (149)Belépési pont: \$E164 (57703)Mód: Mind parancs, mind program módban használható.

Ellenőrzi, hogy valamely periférián elmentett program megegyezik-e a memória megfelelő részén tárolt programmal.

Szintaxis: Megegyezik a LOAD szintaxisával.Példák:

```
10 SAVE "PROG",8
20 VERIFY "PROG",8
100 SAVE "PROG" : REM CSEVELD VISSZA
110 VERIFY
```

READY.

Ugy véljük, a példák önmagukért beszélnek.

Végrehajtás: A VERIFY-t végrehajtó rutin nagyon hasonló a LOAD rutinhoz, azzal a különbséggel, hogy az egyes byte-ok tárolása helyett, a megfelelő memóriarésszel való összehasonlítás történik. A két funkció megkülönböztetésére a ϕ . lapon levő \$A(10) című memória szolgál (ϕ =LOAD, 1=VERIFY).

Hibalehetőségek: Amikor programból hajtjuk végre a VERIFY parancsot, és az OK-val végződik, a program fut tovább. Ha hibát észlel az interpreter ?VERIFY ERROR hibajelzést kapunk és a program futása megszakad.

WAITRövidítés: wA

Token: \$92 (146)

Belépési pont: \$B82C (47148)Mód: Mind parancs, mind program-módban használható.

Az utasítás felfüggeszti a program végrehajtását, míg a paramétereiben specifikált esemény be nem következik.

Szintaxis: WAIT<aritmetikai kifejezés> , <aritmetikai kifejezés> [<aritmetikai kifejezés>]. Az első aritmetikai kifejezés a memória valamely címét jelenti, és így értékének a 0-65535 intervallumba kell esnie. A másik két aritmetikai kifejezés értéke egy-egy byte-os egész szám lehet, így 0-255.

A WAIT L, M1, M2 utasítás először kiszámítja a

(PEEK (L) EOR M2) AND M1

értékét. Ha az eredmény nullától különböző, a BASIC program folytatódik; ha 0, akkor az interpreter a fenti műveletet ismétli. L-nek tehát olyan címnek kell lennie amit a hardver megszakító rutin, vagy a perifériák használnak. /Ha M2-t nem adtuk meg, akkor csak PEEK (L) AND M1 hajtódik végre/.

Az az esemény, amelyik a BASIC program futásának folytatását eredményezi az L cím valamely bitjének /bitjeinek/ magasra és/vagy alacsonyra állítódása lehet. Az M1-ben magasra kell állítani azokat a biteket, amelyek értékére kíváncsiak vagyunk, a többi pedig alacsonyra. M2-ben azokat a biteket kell magasra állítani, amelyek értékének alacsonyra kell válnia a futás folytatásához. Ha például M1=00011000=24 és M2=00010000=16 akkor a program addig áll, míg az L cím 4. bitje alacsony és az 5. bitje magas. Mihelyt a 4. bit magas vagy az 5. bit alacsony lesz, a program fut tovább.

Példák:

WAIT 1,32,32

WAIT 553273,6,6

Az első példánkban a program addig várakozik, míg a kazet-tás egységen valamilyen billentyűt le nem nyomunk. Második példánkban addig vár a program, míg egy sprite a háttérrel nem ütközik./Ehhez persze az kell, hogy a hardver megszakító rutin mozgassa a sprite-ot./

Végrehajtás: A rutin kiszámítja a paramétereket, majd az előbb már vázolt ellenőrzési eljárás kezdődik meg. Ha a várt esemény nem következik be végtelen ciklust kapunk. Ebből a [STOP] billentyű lenyomva tartásával, majd a [RESTORE] lenyomásával léphetünk ki.

Hibalehetőségek: nincs.

4.2 SIMONS' BASIC utasítások

A következőkben röviden összefoglaljuk a C-64 egyik legnépszerűbb BASIC kiterjesztésének, a SIMONS' BASIC-nek az utasításait. Az I/O egységekkel, a grafikus lehetőségekkel és a hanggenerálással kapcsolatos utasítások a 6., 7. illetve 8. Fejezetben kerülnek ismertetésre. A SIMONS' BASIC alapszavait a következő részekre osztva ismertetjük:

- programszerkesztő utasítások,
- sztring műveletek,
- input utasítások /a billentyűzetről/,
- aritmetikai függvények,
- lemezegységre vonatkozó utasítások,
- vezérlésátadó utasítások,
- hibakezelő utasítások.

Programszerkesztő utasítások

KEY <sorszám>, <sztring>

Az első paraméterként szereplő sorszámú F gomb megnyomása az utasítás végrehajtását követően megegyezik a <sztring> -ben szereplő karakterek begépelésével. Összesen 16 F gomb létezik, ezek billentyűzése a következő:

[C= _Fi]

az F(i+8) gombnak számít. F12 billentyűzése tehát [C= _F4] .

DISPLAY

Kilistázza, hogy az egyes F billentyűkhöz milyen sztringek tartoznak.

AUTO <kezdőszám> , <növekmény>

Automatikus sorszámozást biztosít programozáshoz. A [RETURN] megnyomása után a programsort tárolja, majd a sor elejére kiírja a

következő sorszámot. Az automatikus sorszámozást egy üres sor bevitelével leállíthatjuk.

RENUMBER <kezdőszám> , <növekmény>

Ujraszámozza a programot.

PAUSE [<sztring> ,]<aritmetikai kifejezés>

Az utasítás hatására a <sztring> kiíródik a képernyőre és a program futása az <aritmetikai kifejezés> értékének megfelelő ideig felfüggesztődik. Az időt másodpercekben kell megadni. A [RETURN] lenyomásával a program futása az idő letelte előtt folytatható.

RESET <aritmetikai kifejezés>

Az utasítás a DATA mutatót a paraméterben specifikált sorra állítja.

MERGE "<név>" , <egységszám>

A MERGE paramétereinek jelentése azonos a LOAD utasítás paramétereivel. Az utasítás a <név> nevű file-ban levő programsorokat beszúrja a C-64 memóriájában tárolt program sorai közé.

PAGE n

Az utasítás megadja, hogy a LIST hányasával listázza ki a programsorokat. n értékét 1-nél nagyobbra állítva a LIST hatására csak az első n programsor jelenik meg a képernyőn. A [RETURN] lenyomása n-esével folytatja a listázást. PAGE 0 a normális listázást állítja vissza.

OPTION n

Ha n értéke 10, az ezt követő listázások a SIMONS' BASIC alapszavait inverz formában írják ki a képernyőre vagy a nyomtatóra.

Ha a paraméter értéke ettől eltérő ($0 \leq n \leq 255$) akkor a listázás normális./A C-64 eredeti BASIC alapszavai nem inverz alakban íródnak ki!/

DELAY n

A listázás sebességét állítja. $0 \leq n \leq 255$ lehet. A [SHIFT]bíl-lentyű lenyomásával hatása megszűnik.

FIND <sztring>

Kilistázza a program azon sorszámaival, amiben a <sztring> szerepel.

TRACE n

Ha n értéke 10, a program futása közben a képernyőn egy 'ablak' jelenik meg, amiben az éppen végrehajtott utasítások sorszáma látszik, egyszerre maximum 6.

RETRACE

A program szerkesztése után újból kijelzi az utoljára végrehajtott utasítások sorszámát.

DUMP

Kiírja a memóriában tárolt változók értékeit

<változónév> = <érték>

alakban.

COLD

SIMONS' BASIC hideg-indítása. Hatására a C-64 újra végrehajtja a teljes reset ciklust.

C-64

4.113

DISAPA:

Ezzel az utasítással kezdődő sorok a SECURE Ø végrehajtása után nem lesznek listázhatók.

SECURE Ø

Az utasítás végrehajtása után a DISAPA:-val kezdődő sorok nem látszanak a listázáskor. Csak a megfelelő memóriarekesz POKE-olásával hatástalanítható:

```
Például      1Ø DISAPA : PRINT "EZ ITT VAN"
              SECURE Ø : LIST
```

hatására csak

1Ø

jelenik meg a képernyőn.

OLD

Hatástalanítja a NEW utasítást.

Sztring műveletek

INSERT (<sztring> , <sztring> , <aritmetikai kifejezés>)

A harmadik paraméterként megadott sorszámú karaktertől kezdődően a második <sztring> -be beszúrja az első <sztring> -et.

INST (<sztring> , <sztring> , <aritmetikai kifejezés>)

A második paraméterként szereplő <sztring> karaktereit az <aritmetikai kifejezés> -ben megadott helytől kezdődően az első <sztring> -re cseréljük.

PLACE (<sztring>, <sztring>)

Az interpreter ellenőrzi, hogy az első paraméterként szereplő <sztring> előfordul-e a második <sztring>-ben. Ha igen, a függvény értéke az első előfordulás első karakterhelyére mutat. Ha \emptyset , akkor az első <sztring> nem fordul elő a második <sztring>-ben.

DUP (<sztring>, <aritmetikai kifejezés>)

A <sztring> értékét a második paraméter értékének megfelelő sokszor adja össze.

CENTRE <sztring>

Az utasítás a következő sor közepére kiírja a <sztring> -nek megfelelő szöveget.

AT (<aritmetikai kifejezés>, <aritmetikai kifejezés>)

A PRINT utasításban elválasztóként a TAB, SPC stb. mellett az AT utasítás is szerepelhet. Hatására a nyomtatás az AT-ben specifikált karakterhelytől folytatódik.

USE <nyomtatási kép>, <változó> : PRINT

Az utasítás hatására a <változó> sztring kifejezés a <nyomtatási képnek> megfelelő formában kerül kiírásra. Ha a PRINT után szerepel a ; vagy a , nem kerül sor automatikus kocsi-vissza-soremelés kiírására.

A <nyomtatási kép> valójában egy sztring, amelyben szereplő # és . karaktereknek speciális jelentése van. A # jel a kiírandó sztring tizedespont előtti, illetve utáni számjegyeit jelenti. Például

```
USE "###.###", X$: PRINT
```

a számot úgy írja ki, hogy az egész rész 3, a törtrész 5 karakterhelyet foglaljon el. A sztringben további karakterek is lehetnek, ezek a megfelelő helyen kiíródnak. Például

```
USE "$####", X$: PRINT
```

hatása X=2, 123, 1544 illetve 12331 és X\$=STR\$(X) esetén rendre a következő:

```
$ 2
$ 123
$1544
$2331
```

A USE "###.###", X\$: PRINT az X\$="1236.58" értékét
1 2 3 6 . 6

alakban írja ki.

Input utasítások

FETCH "<sztring>", <aritmetikai kifejezés>, <változó>

Az utasítás hatása végsősoron megegyezik az INPUT <változó> hatásával; a <sztring> és <aritmetikai kifejezés> értékének megadásával azonban mód van az INPUT ellenőrzésére. A FETCH nem ad ?-jelet és a kurzor nem villog. A <sztring> a beírható karakterek típusát jelenti:

<sztring>	<u>beírható karakterek</u>
"[Home]"	betük
"[CRSR↓]"	betük kivételével minden
"ABCDE"	csak az A,B,C,D,E karakterek

Az <aritmetikai kifejezés> az inputsor maximális hosszát adja meg. Ennél több karaktert az interpreter nem fogad el. Az inputsor bevitelét a [RETURN] megnyomásával kell befejezni.

<változó> = INKEY

Az értékadás eredménye a benyomott F gomb indexe lesz /például 3, ha F3-at nyomtuk meg/.

ON KEY <sztring>, : GOTO <sorszám>

Az utasítás ellenőrzi, hogy valamely a <sztring>-ben szereplő karaktereknek megfelelő billentyű meg van-e nyomva. Ha igen, a vezérlésátadás végrehajtódik. Például a

```
300 ON KEY "1234567890", : GOTO 400
310 GOTO 300
```

programrész ciklusban várakozik addig, míg egy számjegy billentyűt le nem nyomunk. /Lásd még az alábbi két utasítást is!/

DISABLE

Az ON KEY utasítás az interpreter megszakító rendszerébe avatkozik bele, ezért mihely a megfelelő billentyűt megnyomtuk, célszerű ezt a hatást megszüntetni. Erre szolgál a DISABLE utasítás. A fenti példa esetében a 400. sor általában egy DISABLE utasítással kell, hogy kezdődjön.

RESUME

Feltétel nélküli vezérlésátadás az utoljára végrehajtott ON KEY utasításra.

Aritmetikai függvények

MOD(X,Y)

A függvény elosztja X-et Y-nal és a maradékkal tér vissza.

DIV (X,Y)

A legnagyobb olyan egész számmal tér vissza, amelyet Y-nal megszorozva még nem kapunk X-nél nagyobb számot. $\text{INT}(X/Y)$ /

FRAC (X)

X törtrésze $\text{INT}(X) /$.

A SIMONS' BASIC lehetővé teszi bináris és hexadecimális konstansok használatát. Ezek 8 és 4 bináris, illetve hexadecimális jegyből kell, hogy álljanak. Például

```

%00001011  /= 11/
$00AF       /= 175/

```

EXOR (X,Y)

A 2-byte-os számmá konvertált X és Y bitekre bitenként végrehajtja a 'kizáró vagy' logikai műveletet.

Lezemegeységre vonatkozó utasításokDISK <parancs>

Az utasítás hatása ekvivalens a következővel

```

OPEN 15,8,15
PRINT 15,<parancs>
CLOSE 15

```

DIR "\$ vagy DIR "\$:<szöveg>

Az utasítás hatására a lemez katalógusa vagy annak a szöveg - gel egyező nevű file-ja a képernyőre kerül. A ? és a * jelek használata a szokásos /lásd az 5. fejezetben/.

Vezérlésátadó utasításokCGOTO <aritmetikai kifejezés>

Kiszámított GOTO utasítás. Az <aritmetikai kifejezés> értékének megfelelő sorszámú sorral folytatódik a program végrehajtása.

IF <logikai kifejezés> THEN <utasítás> : ELSE: <utasítás>

Az ELSE alapszóval bővített feltételes vezérlésátadó utasítás lehetővé teszi, hogy a <logikai kifejezés> = hamis esetben a vezérlés ne a következő programsorra, hanem az ELSE-t követő utasításra kerüljön át.

REPEAT ... UNTIL

Az utasításpár egy másfajta ciklusszervezési lehetőséget biztosít, mint a FOR...NEXT. A REPEAT utasítás alakja:

REPEAT <utasítás>

Az UNTIL alakja:

UNTIL <logikai kifejezés>

A REPEAT utasítás végrehajtásakor a verembe kerül a visszatérést biztosító CHRGET mutató és a REPEAT tokenje; majd a REPEAT-et követő <utasítás> hajtódik végre. Az UNTIL végrehajtása a <logikai kifejezés> kiértékelésével kezdődik. Ha igaz, a program végrehajtása folytatódik, ha nem, a vezérlés az utoljára végrehajtott REPEAT-ben szereplő <utasítás> végrehajtásával folytatódik. Ezt az interpreter úgy éri el, hogy a vermet végignézi, megkeresi a REPEAT tokent és visszaállítja a CHRGET mutatóját.

RCOMP: <utasítás> : [ELSE :<utasítás>]

A legutóbbi IF ...THEN...ELSE utasítás feltételét értékeli ki. Ha igaz, az első utasítás hajtódik végre, ha nem, a második. Ha hiányzik, a vezérlés hamis feltétel esetén a következő programsorra kerül.

LOOP...EXIT IF...END LOOP

A három egymással szorosan összefüggő utasítás lehetőséget biztosít a ciklusból bármelyik pontján történő kilépésre. Az egyes utasítások alakja a következő:

```

LOOP <utasítás>
EXIT IF <logikai kifejezés>
END LOOP

```

A LOOP utasítás hatására a verembe kerül a visszatérést biztosító CHRGET mutató, a programsor száma és a LOOP tokenje, majd végrehajtódik az utasítás.

Az EXIT IF utasítás végrehajtása a <logikai kifejezés> kiértékelésével kezdődik. Ha hamis, a következő utasításra kerül a vezérlés. Ha igaz, az interpreter az utasítás sorszámánál nagyobb sorszámú programsorokban elkezd az END LOOP utasítás keresését. Az első END LOOP utáni utasításra kerül a vezérlés.

Az END LOOP utasítás az utoljára végrehajtott LOOP-ban levő <utasítás>-ra adja a vezérlést. Ehhez az interpreter végignézi a vermet és megkeresi az első LOOP tokenet. Ezután a szükséges értékeket visszaállítja, a CHRGET mutatóját a LOOP utánra állítja.

PROC <név>

Az utasítás az utána kezdődő programrészt a <név>-vel jelöli meg. A SIMONS' BASIC két alábbi utasításával lehet az így kapott címkékre hivatkozni:

CALL <név>

Feltétel nélküli vezérlésátadás a <név> címkével megjelölt programrészre.

EXEC <név>

A <név> címkéjű alprogram végrehajtása. A verembe kerül a programsor száma, a CHRGET mutató és az EXEC tokenja. Az alprogramot nem a RETURN hanem az alábbi utasítással kell befejezni:

END PROC

Az utasítás megkeresi a veremben található első EXEC token-t és az ott talált adatoknak megfelelő helyről folytatja a program végrehajtását. /Nem helyettesíthető a RETURN utasítással./

LOCAL <változólista>

Az interpreter elmenti a <változólista>-ban szereplő változók értékét a legközelebbi GLOBAL utasításig. Addig a változólistában a szereplő változókat - más célokra - tovább használhatjuk.

GLOBAL

A legutolsó LOCAL utasításban használt változók elmentett értékeit visszatölti. A LOCAL-GLOBAL utasításpárt általában az alábbi környezetben használjuk:

```
PROC <név>
LOCAL <változólista>
...
<az eljárás törzse>
GLOBAL
END PROC
```

Hibakezelő utasítások

A C-64 interpreter úgy lett megtervezve, hogy hiba esetén a program futása szakadjon meg és a hibaüzenet kerüljön kiírásra. Néha azonban hasznos, hogy a programon belül is kezelni tudjuk a felmerülő hibákat. Erre szolgálnak az alábbi utasítások.

ON ERROR: <utasítás>

Az utasítás végrehajtása annyit jelent, hogy az interpreter tudomásul veszi, hogy hiba esetén az <utasítás>-t kell a programnak végrehajtania.

ERRN

Változó, ami hiba esetén a hiba számát /kódját/ tartalmazza.

ERRL

Változó, ami hiba esetén a hibát okozó programsor számát tartalmazza.

NO ERROR

Az utasítás megszünteti a hibakezelő rutin hatását és ezután hiba esetén a program futása megszakad.

OUT

Az utasítás a C-64 hibaüzeneteit kiírja és megszakítja a program futását, azonban az ON ERROR hatását nem szünteti meg. Például GOTO-val folytatva a programot a futás közben a hibakezelés az ON ERROR-nak megfelelően történik.


```

10 *=$033C          46 STA $D800,X
20 LDX #$05          50 DEX
30 LP LDA $61,X      60 BPL LP
40 STA $0400,X       70 RTS
45 LDA #$0D

```

Egyetlen példánk az #1 lebegőpontos akkumulátor tartalmát közvetlenül a képernyő első hat karakterpozíciójának helyére írja ki. Ilyen módon lehetőség nyílik annak tanulmányozására, hogyan tárolja a gép az egyes számokat.

USR(0) eredménye	000000	vagyis 0 0 0 0 0 0
USR(1) eredménye	129000	vagyis 129 128 0 0 0 0

inverz A@

Végrehajtás: A rutin kiszámítja és ellenőrzi az argumentum értékét, majd az #1 lebegőpontos akkumulátorba helyezi. Ezután egy JMP \$0310 utasítás kerül végrehajtásra. A RTS utasítás végrehajtása után a rutin folytatja a BASIC program végrehajtását. Amennyiben a USR függvénnnyel értéket is szeretnénk visszaadni /például ?USR(1) + USR(2)/ akkor ezt az értéket az #1 lebegőpontos akkumulátorban kell hagynunk.

Hibalehetőségek: A BASIC interpreter nem ellenőrzi a gépi-kódu program futását. A USR használatára ugyanaz vonatkozik mint a SYS-re.

VAL

Rövidítés: VA

Token: \$C5 (197)

Belépési pont: \$B7AD (47021)

Mód: Mind parancs, mind program módban használható.

Az argumentumaként szereplő sztring kifejezés numerikus értéket számítja ki. A sztringet első - szám részeként már nem értelmezhető - karakteréig dolgozza fel.

5. Fejezet

LEMEZEGYSÉG

5.1 Bevezetés

Ebben a fejezetben a C-64-hez csatlakoztatható VIC-1541 típusu lemezegység használatát ismertetjük. A C-64-hez a Commodore cég ma már 'kemény' vagy más néven Winchester lemezeket is forgalmaz, mégis leggyakrabban háttértárolóként hajlékonylemez /floppy/ tárolót használnak. A C-64-hez lehetőség van még kazettás egység csatlakoztatására is, ennek a lassúsága és megbízhatósága azonban még a legigénytelenebbeket sem elégitheti ki.

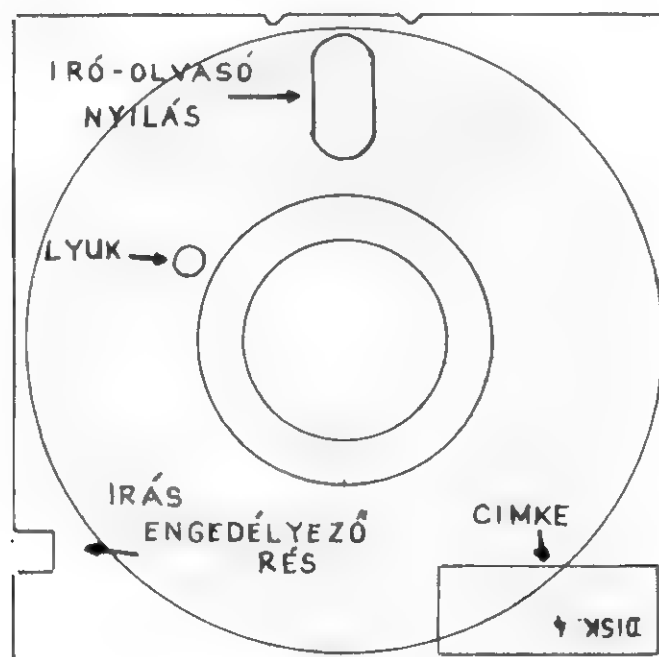
A legtöbb lemezegység - így a VIC-1541 is - egy író/olvasó, illetve egy törlő fejjel rendelkezik, amelyeket egy léptető motor segítségével lehet a lemez megfelelő sávja fölé pozicionálni. A motor mindig sugárirányban mozgatja a fejeket, egy-egy lépés nagysága megközelítőleg 1/40 inch. A lemezen annak a sávnak a szélessége, amelyen az írás végül is történik, körülbelül 1/80 inch. A lemezegység motorja nagy sebességgel forgatja a lemezt, amelyik a centrifugális erő hatására elveszti hajlékony jellegét, és így lehetővé válik az író/olvasó fej mozgatása.

A lemezegység számos áramkört, érzékelőt tartalmaz, amelyek lehetővé teszik a léptető motor vezérlését, az írást engedélyező rés meglétének ellenőrzését, elolvassa a lemezegység adott sávját stb. A VIC-1541 lemezegység egy önálló 16K-s operációs rendszert /DOS/ tartalmaz, amelyik a lemezegység kezelésén túlmenően a C-64-gyel való kommunikációt is biztosítja. A DOS használata nagyfoku önállóságot eredményez, ami lehetővé teszi, hogy a központi egységtől függetlenül is dolgozhasson. Elérhető

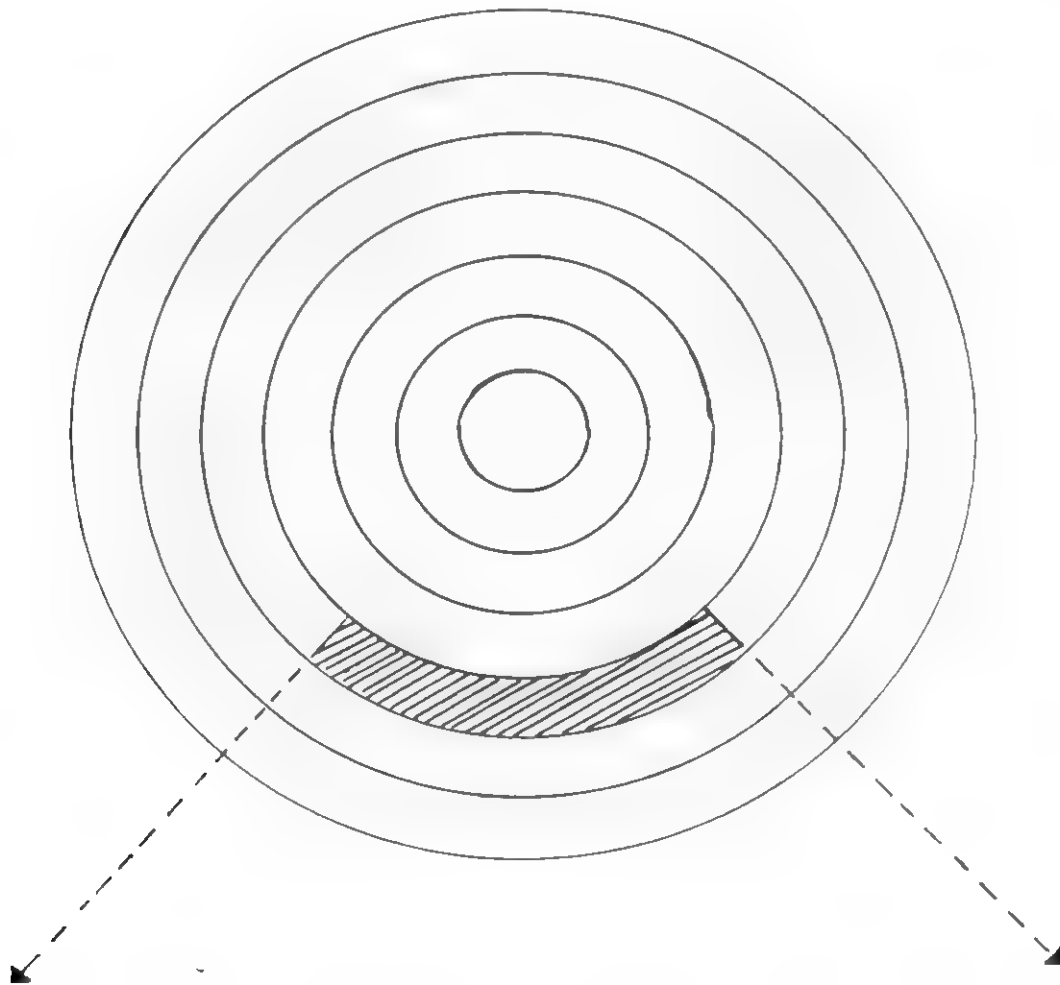
például, hogy a lemezegység egy file-t a sornyomtatón listázzon, miközben a C-64 valami más feladatot végez.

A C-64 lemezegység kimenete /SERIAL PORT/ úgy lett megtervezve, hogy egyszerre 5 lemezegység és egy nyomtató legyen rácsatlakoztatható. Ebben az esetben természetesen a lemezegységek számát /ami - a priori - 8/ módosítani kell.

Az alábbi ábra megközelítőleg jól ábrázol egy lemezt. A négyzet a lemezt magában foglaló borítékot szemlélteti, a kör alakú rész a mágneses adathordozó valódi /de nem látható/ szélét mutatja. Az írásvédő rés lehetővé teszi a lemezek fizikai védelmét. Amennyiben az írásvédő részt kivágtuk, a DOS nem akadályozza meg a lemezre írást. Ha a rés hiányzik /pl. leragasztottuk/, a DOS nem engedi meg a lemezre való írást.



A DOS a lemezre sávonként /track/ írja az információt. A sávok koncentrikus körgyűrűk. A lemezen 35 sáv helyezkedik el, amelyeket kívülről 1-gyel kezdődően számoz meg a rendszer. A sávra szeletenként, szektoronként /sector/ egy blokknyi adat /256 byte/ információ kerül felírásra úgy, ahogy ez az alábbi ábrán látható:



Egy szektorba a következő információk kerülnek rögzítésre:

SYNC	08	ID1	ID2	TRACK	SECTOR	ELL. ÖSSZEG	GAP1	SYNC
	07	256 byte adat			ELL. ÖSSZEG	GAP2		

A DOS rendszer automatikusan gondoskodik ezeknek az elhelyezéséről. A programozó feladata csupán a 256 adatbyte-nak a puffokban való elhelyezése. A lemezegység - eltérően a legtöbb gyártótól - a külső /alacsonyabb sorszámú/ sávokba sűrűbben ír, és így ott több szektor fér el:

<u>Sáv /track/ sorszáma</u>	<u>A szektorok indexelése</u>	<u>Összes szektor</u>
1-17	0-20	21
18-24	0-18	19
25-30	0-17	18
31-35	0-16	17

A fenti adatokból is látható, hogy a lemezre összesen 683 blokknyi információ fér el. Ezek közül a DOS 19 blokkot szervezési célokra használ, ezért egy lemez tárolókapacitása 168566 byte /664 blokk/.

A lemezegység és a lemezek megfelelő karbantartása esetén egy bit meghibásodásának a valószínűsége igen kicsiny $\approx 10^{-11}$. A DOS rendszerhibák ennél gyakoribbak. A DOS, ha hibát észlel, az író/olvasó fejet a lemez széléig lépteti, majd újból megkísérli az olvasást. Ezt az eljárást tízszer ismétli, és csak azután ad hibajelzést.

5.2 Az adatok tárolása

A DOS a lemezen tárolt adatokról nyilvántartást vezet, ami rögzíti, hogy a lemez mely sávjai és szektorai tartalmaznak információt; milyen programokat, adatfile-okat tárolunk a lemezen stb. A DOS két módot kínál a lemezre való írásra/olvasásra. Az első esetben a lemez katalógusában /directory/ szereplő file-okat használjuk, a második esetben közvetlenül az általunk kiválasztott blokkból írunk /vagy onnan olvasunk ki/ információt.

Ebben a részben azokat a legfontosabb fogalmakat ismertetjük, amelyek az adattárolás megértéséhez feltétlenül szükségesek.

Szekvenciális file A szekvenciális file-ok lemezen és kazettán egyformán használhatók. A file adatait csak egymás után - a kiírás sorrendjében - tudjuk visszaolvasni /ami a szalagon természetes/. A lemezegységen tárolt szekvenciális file használata azonban lényegesen gyorsabb, s ismételt használatához nem kell újra csévélni a szalagot. A kazettás egységen egyszerre csak egy file lehet megnyitva, míg a lemezes egységen maximum 10.

A szekvenciális file adatbyte-ok egymásutáni sorozatától mindössze annyiban tér el, hogy rekordokra oszlik. Egy-egy rekord végét a CHR\$(13) byte jelzi. Az INPUT# utasítás egyszerre egy rekordot olvas be a lemezről, míg a GET# utasítás karakterenként /byte-onként olvas/, beleértve a rekordok végét jelző CHR\$(13) karaktert is. A szekvenciális file végét - ha nem használtunk speciális jelzőket - az ST állapotváltozó nullától eltérő értéke jelzi /lásd ST-t a 4. fejezetben!/. A PRINT# utasítás, amikor egy sor feldolgozását befejezte, automatikusan küld egy CHR\$(13) karaktert még a lemezre, így lehetőség van az INPUT# használatára, amelyik input-sornak a következő rekordot tekinti. Ha az inputsor tartalmaz vesszőt /,/ vagy kettőspontot /:/ az adatok elkülönítésére, akkor az INPUT# X,X\$,Y\$,Z\$ típusu utasítások segítségével lehetőségünk van ezeket külön-külön is visszaolvasni.

Amikor például a PRINT#8,X\$; CHR\$(13); utasítást többször is használjuk, akkor egy ilyen eredményt kapunk:

Byte:	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8
File:	K	U	T	Y	A	[R]	M	A	C	S	K	A	[R]	E	G	E	R	[R]

←────────→	←────────→	←────────→
1. rekord	2. rekord	3. rekord

Program file A lemezegység lehetőséget biztosít a C-64 programjainak SAVE utasítással való tárolására, ami azután a LOAD utasítás segítségével visszatölthető. A program file-ok speciális szekvenciális file-ok és szekvenciális file-ként is olvashatók. A file első két byte-ja azt a memóriahelyet tartalmazza, ahová a programot be kell tölteni, ezután következik maga a program abban a formában, ahogy a memóriából kiírtuk.

Katalógus /Directory/ A DOS a lemezen tárolt file-ok kezdéséhez szükséges információt a 18. sáv 0. szektorában kezdődő katalógusban tárolja. Ezek az információk a következők:

- /i/ a lemez neve és azonosító száma;
- /ii/ a lemez formája;
- /iii/ a lemezen levő file-ok neve, típusa és hogy hány blokkot foglalnak el külön-külön a lemezen;
- /iv/ melyek a foglalt blokkok.

A katalógusra a különböző utasításokban a \$ jellel lehet hivatkozni. A katalógus egy speciális program file-nak tekinthető, amely például a LOAD "\$",8 utasítással betölthető a memóriába és kilistázható. A /iv/ alatti információ azonban nem képezi részét ennek a file-nak.

A blokk foglaltság jelzésére 140 byte szolgál a katalógus elején. A katalógusnak ezt a részét BAM-nak hívjuk az angol elnevezés /block availability map/ alapján. A következő /5.7/ oldalon a katalógus első blokkját listáztuk ki, amelyen jól látható, hogyan helyezkednek el a fenti részek a lemezen.

Mint látható, a blokk első két száma mindig a file következő blokkjának sáv- illetve szektorszámát adja meg. /Nemcsak a katalógus van így szervezve, hanem valamennyi szekvenciális- és program file/. Mivel 0 sorszámú sáv nincs, ezért a 0 sáv a file utolsó blokkját jelenti. Ebben az esetben a 2. byte az adott

18. sáv 0. szektor:

következő sáv, szektor

BAM

12 01	41 00	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	:	A	ππ	ππ	ππ
15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	:	ππ	ππ	ππ	ππ
15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	15 FF FF 1F	:	ππ	ππ	ππ	ππ
15 FF FF 1F	15 FF FF 1F	11 D7 5F	1F 00 00 00	00 00 00 00	10 EC FF 07	:	ππ	ππ	04	
00 00 00 00	00 00 00 00	0A B0 C3	07 13 FF FF	07 13 FF FF	07 13 FF FF	:		ππ	ππ	
00 00 00 00	0A B0 C3	07 13 FF FF	07 13 FF FF	07 13 FF FF	07 13 FF FF	:		ππ	ππ	
13 FF FF 07	12 FF FF 03	12 FF FF 03	12 FF FF 03	12 FF FF 03	11 FF FF 01	:	ππ	ππ	ππ	ππ
12 FF FF 03	12 FF FF 03	12 FF FF 03	12 FF FF 03	11 FF FF 01	11 FF FF 01	:	ππ	ππ	ππ	ππ
11 FF FF 01	11 FF FF 01	11 FF FF 01	11 FF FF 01	11 FF FF 01	11 FF FF 01	:	ππ	ππ	ππ	ππ
31 35 34 31	54 45 53 54	2F 44 45 4D	4F A0 A0 A0	A0 A0 A0	A0 A0 A0	:	1541	TEST/DEMO		
A0 A0 5A 58	A0 32 41	A0 A0 A0	A0 A0 A0	00 00 00	00 00 00	:	ZX	2A		
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:				
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:				
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:				
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:				
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:				

file típus (PRG)

következő sáv, szektor

a file mérete blokkokban

a file neve (16 karakter)

12 04	82 11 00	48 4F 57 20	54 4F 20 55	53 45 A0	:	II	HOW TO USE
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:	II	HOW PART TW
00 00 82 11	03 48 4F 57	20 50 41 52	54 20 54 57	4F A0 A0 A0	:	II	VIC-20 WEDG
00 00 82 11	09 56 49 43	2D 32 30 20	57 45 44 47	45 A0 A0 A0	:	E	C-64 WEDGE
00 00 82 13	00 43 2D 36	34 20 57 45	44 47 45 A0	A0 A0 A0 A0	:	II	C-64 WEDGE
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:		

a katalogus utolsó blokkja

00 FF	82 10 01	44 49 52 A0	A0 A0 A0	A0 A0 A0	:	ππ	DIR
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:	II	VIEW BAM
00 00 82 10	03 56 49 45	57 20 42 41	4D A0 A0 A0	A0 A0 A0	:	II	CHECK DISK
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:	II	DISPLAY T&S
00 00 82 10	0F 44 49 53	50 4C 41 59	20 54 26 53	A0 A0 A0 A0	:	II	PERFORMANCE
00 00 82 14	02 50 45 52	46 4F 52 4D	41 4E 43 45	20 54 45 53	:	TEST	
00 00 82 14	07 53 45 51	55 45 4E 54	49 41 4C 20	46 49 4C 45	:	II	SEQUENTIAL
00 00 82 0F	01 52 41 4E	44 4F 4D 20	46 49 41 4C	A0 A0 A0 A0	:	II	RANDOM FIAL
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:	II	FILE COPY
00 00 82 15	00 46 49 4C	45 20 43 4F	50 59 A0 A0	A0 A0 A0 A0	:		
A0 A0 A0 A0	A0 A0 00 00	00 00 00 00	00 00 00 00	00 00 00 00	:		

blokkon belül az utolsó, még a file-ba tartozó byte sorszámát jelenti.

Egy-egy szektor foglaltságát 4 byte jelzi. Az első byte a sáv /track/ szabad byte-jainak számát adja meg, az azt követő 24 bit pedig a sávok foglaltságát jelzi. 0 byte foglaltat, az 1 szabadot jelent. A nem létező szektoroknak megfelelő utolsó bitek mindig nullák.

Véletlen file Mint már utaltunk rá, lehetőségünk van egy tetszőleges blokk tartalmát a lemezegység pufferébe olvasni, illetve a puffer tartalmát egy tetszőleges blokkba beírni. /Ez akár a 18. sáv 0. szektora is lehet, ami azt jelenti, hogy a BAM-ot is át tudjuk írni./ Ennek a lehetőségnek a használatát a # nevű pszeudo-file megnyitásával érhetjük el. A * file valójában a lemezegység pufferét jelenti, és a szokásos író/olvasó utasításokon kívül további utasításokat kell kiadnunk, ami a puffer és a lemez megfelelő blokkja közti adatátvitelt biztosítja. További lehetőséget biztosít a relativ file-ok használata.

5.3 A 15. csatorna használata

A lemezegység 15-ös csatornája, az ugynevezett parancs vagy hiba csatorna speciális funkciót tölt be. Ezen keresztül adhatunk közvetlen utasításokat a DOS-nak, illetve olvashatjuk el a DOS hibaüzeneteit. A csatornát általában az

```
OPEN 15,8,15
```

utasítással nyitjuk meg.

Lemezek formázása Valahányszor egy új, még addig nem használt lemezt akarunk használni, a lemezt először formázni kell. A DOS ilyenkor elhelyezi a megfelelő szinkronizációs jeleket, felírja az üres katalógust stb. Az utasítás alakja:

```
PRINT#15, "NEW : <név>, id"
```

vagy rövidítve:

```
PRINT #15, "N :<név>, id"
```

A <név> karaktersorozat a lemez neve lesz. A név csak a katalógusba kerül beírásra, míg az id két karakteres azonosító valamennyi blokkba. Ha két kiíró utasítás között kicseréljük a lemezt, a DOS érzékeli, hogy új id azonosítóju lemezzel dolgozik. A fenti utasítás

```
PRINT #15, "N :<név>"
```

alakja csak a katalógust törli, de nem formázza újra a lemezt, és csak a nevet változtatja meg.

A NEW DOS-parancs természetesen használt lemezekre is kiadható, de ebben az esetben a rajta levő információ teljes egészében elvész. Használata előtt tehát mindig győződjünk meg róla, hogy a rajta levő adatokra már nincs szükségünk.

A lemezegység inicializálása Elsősorban DOS rendszerhiba után előfordulhat, hogy a lemezegység pufferében tárolt BAM nem egyezik meg a lemezen levő BAM-mal. Ez meglevő file-ok felülírását eredményezheti. Ilyen esetben célszerű a lemezegységet inicializálni, melynek hatására a DOS újra olvassa a BAM-ot. Az utasítás ekvivalens a lemez kivételével, a lemezegység ki majd bekapcsolásával, a lemez visszahelyezésével, végül a 15. csatorna újbóli megnyitásával. Az utasítás alakja:

```
PRINT #15, "INITIALIZE" vagy rövidítve:
```

```
PRINT #15, "I"
```

File-ok törlése Lehetőség van egy vagy több file törlésére. A törlés nem jelenti a fizikai törlést, csupán a katalógusban jelzi egy bit, hogy a szóban forgó file már nem létezik. Amikor legközelebb egy file-t hozunk létre, az írja felül a lemez

megfelelő részeit. Az utasítás alakja:

```
PRINT #15, "SCRATCH :<név>" vagy
PRINT #15, "S :<név>"
```

A névben szereplő * illetve ? karakterek jelentése speciális. A * karakter csak a név utolsó karaktere lehet. Ebben az esetben az összes, azzal a névvel kezdődő file törlődik. Ha a

```
PRINT #15, "S : GRID*"
```

utasítást adjuk ki, akkor törlődnek a következő programok: GRID, GRIDBOOT, GRIDRUNNER stb. A kérdőjel azt jelenti, hogy lényegtelen, azon a helyen milyen betű áll. Ha a katalógus tartalmazza a PEK, POK, PIK nevű programokat, akkor a

```
PRINT #15, "S : P?K"
```

valamennyit törli.

A lemez újraszervezése Ha egy lemezt már sokat használtunk, akkor az egy file-hoz tartozó blokkok össze-vissza helyezkednek el a lemezen, és a DOS igen lassan tudja csak olvasni őket. Lehetnek a lemezen az OPEN utasítással megnyitott, de szabályosan le nem zárt file-ok is. A lemez rendbetételére a

```
PRINT #15, "VALIDATE" vagy rövidítve a
PRINT #15, "V"
```

utasítást használhatjuk. Az utasítás hatására a DOS újraszervezi a lemezeken levő adatokat, anélkül, hogy azok tartalmát megváltoztatná. A VALIDATE DOS-parancs törli a véletlen-file blokkjait, ezért olyan lemez esetén, amelyik tartalmaz véletlen-file-t is, semmiképp se használjuk.

File-ok átnevezése Néha szükség lehet arra, hogy egy file-nak megváltoztassuk a nevét. Ezt a következő utasítással érhetjük el:

```
PRINT #15, "RENAME :<új név>=<régi név>"
PRINT #15, "R :<új név>=<régi név>"
```

File-ok másolása és összefűzése Lehetőségünk van egy file másolatát ugyanazon a lemezen, de más név alatt létrehozni:

```
PRINT #15, "COPY :<uj név> = <régi név>" vagy
PRINT #15, "C :<uj név> = <régi név>"
```

A COPY DOS-parancsot maximum négy file összefűzésére is használhatjuk. A COPY ebben az esetben csak szekvenciális file-okat tud összefűzni:

```
PRINT #15, "COPY :<uj név>=<régi név1>,<régi név2>,<régi név3>,<régi név4> "
```

A hibacsatorna olvasása A 15. csatorna az

```
INPUT #15, A$, B$, C$, D$
```

utasítással olvasható. Az A\$, B\$, C\$, D\$ értékei a DOS hibaüzenetét tartalmazzák. Az egyes változók jelentése:

A\$ = a hiba számkódja;	
B\$ = a hiba megnevezése;	
C\$ = a sáv	} száma, amihez a hiba kapcsolódik.
D\$ = a szektor	

Az A\$, C\$, D\$ csak számjegyeket tartalmaz, ezért az INPUT utasításban aritmetikai változatokkal helyettesíthetjük őket.

5.4 Program file-ok használata

Programok lemezen való tárolására, illetve visszatöltésére speciális utasítások állnak rendelkezésre. A DOS a tárolásra és töltésre a lemezegység 0. illetve 1. csatornáját használja, ezért ezeket adatcsatornaként nem lehet üzemeltetni.

Programok töltése A lemezen tárolt programok töltésére a LOAD utasítás szolgál. Alakja:

```
LOAD <név>, <egységszám> ,<mód>
```

A <név> a program nevét definiáló sztringkifejezés; az <egységszám> a lemezes egység sorszámát adja meg, ami általában 8 vagy 9. A <mód> értéke 0 vagy 1 lehet, /ha nem írjuk ki, az 0-nak felel meg/. <mód>= 1 esetén a program a memóriába pontosan oda íródik vissza, ahonnan a SAVE utasítással elmentettük. <mód>= 0 esetén a program a BASIC munkaterület elejére töltődik be.

A katalógus a LOAD "\$", 8 utasítással tölthető be, majd LIST utasítással olvasható.

A LOAD részletes ismertetése a 4. fejezetben található.

Programok tárolása A memóriában tárolt programokat a SAVE utasítás segítségével menthetjük ki lemezre. Az utasítás formája:

SAVE <név>, <egységszám> ,<mód>

A <név> és <egységszám> jelentése ugyanaz, mint a LOAD utasításban. /A <mód>-értékét a gép semmire sem használja, csupán azért szerepelhet, mert a két utasítás szintaxisát ugyanaz a rutin ellenőrzi./

Az utasítás - feltéve, hogy <név> nevű file még nincs a lemezen - létrehoz a lemezen egy új, <név> nevű file-t és a memóriában tárolt program tartalmát elmenti a file-ba. Ha a file már létezik, a mentés nem történik meg és hibajelzést kapunk. Ezt elkerülendő a SAVE első paraméterét

"Q : <név>"

alakban kell megadnunk. Ennek hatására a DOS először törli a meglévő file-t, majd a SAVE normálisan végrehajtódik.

Programok ellenőrzése A

VERIFY <név>, <egységszám>

utasítás ellenőrzi, hogy a <név> nevű file tartalma megegyez-

zik-e a C-64 memóriájában tárolt programmal. Részletek a 4. fejezetben találhatók.

Példák Szükségünk lehet egy lemezen tárolt program kezdőcímének ismeretére. Mint már említettük, a kezdőcim a megfelelő file első két byte-ja. Ennek megfelelően a következő program egy tetszőleges program file kezdőcímét állítja elő:

```
10 INPUT "FILE NEV";N$
20 OPEN 1,8,2,N$+",PRG,READ"
30 GET#1,X$:IF X$="" THEN X$=CHR$(0)
40 GET#1,Y$:IF Y$="" THEN Y$=CHR$(0)
50 X=ASC(X$):Y=ASC(Y$)
60 PRINT "KEZDOCIM=";X+256*Y
70 CLOSE1
```

Elképzelhető, hogy olyan gépi kódu programunk van, amelyet a SAVE utasítással nem tudunk azonnal kiíratni. Ilyen eset állhat elő, ha BASIC-ből állítjuk elő az utasításokat. Ilyenkor a program file-ba közvetlenül kell írunk. Erre példa a következő programrész, amelyik a BASIC munkaterület elején elhelyezkedő program file-t állítja elő. /A programot töltsük be és indítsuk el!/

```
10 OPEN 1,8,2,"GEPIKOD,PRG,WRITE"
20 PRINT#1,CHR$(1)CHR$(8);
30 PRINT#1,CHR$(162)CHR$(0)CHR$(138)
40 PRINT#1,CHR$(157)CHR$(0)CHR$(4)CHR$(232)CHR$(208);
50 PRINT#1,CHR$(249)CHR$(96)
60 CLOSE1
```

Program file-ok közvetlen írásával és olvasásával elérhetjük programrészek egymáshoz fűzését. A következő oldal elején látható rövid program ezt a feladatot végzi el. A 120-160 sorszámú programsorok a PROG1 file karaktereit átmásolják a PROGEREDO nevű file-ba, kivéve az utolsó két 0 byte-ot. A PROG2 file másolását a két első byte /a kezdőcim/ kihagyásával /190.sor/

kezdjük, majd karakterenként átmásoljuk. A program az APPEND utasítást szimulálja.

```

100 OPEN 2,8,2,"PROG1,P,R"
110 OPEN 3,8,3,"PROGEREDO,P,W"
120 GET#2,X$
130 Y$=X$:GET#2,X$:IF ST<>0 GOTO 170
140 IF Y$="" THEN Y$=CHR$(0)
150 PRINT#3,Y$;
160 GOTO 130
170 CLOSE 2
180 OPEN 4,8,4,"PROG2,P,R"
190 GET#4,Y$:GET#4,Y$
200 GET#4,Y$:IF ST<>0 GOTO 230
210 PRINT#3,Y$;
220 GOTO 200
230 PRINT#3,CHR$(0);
240 CLOSE 3: CLOSE 4

```

Következő példánk olyan programot mutat be, ami nem csak a programok összefűzését végzi el, hanem át is számozza a programsorok elején levő mutatókat.

```

5 INPUT "ELSO FILE NEVE=";E$
6 INPUT "MASODIK FILE NEVE=";M$
7 INPUT "EREDO FILE NEVE=";F$
10 OPEN 2,8,2,E$+",P,R":OPEN 3,8,3,F$+",P,W"
20 GOSUB 900,ME=NE:GOSUB 1100:EL=0:MUT=NE
30 PRINT"KEZDOCIM=";NE:GOSUB 2000
40 CLOSE 2:OPEN 2,8,2,M$+",P,R"
50 GOSUB 900:EL=MUT-NE:MUT=NE:GOSUB 2000
60 PRINT#3,CHR$(0);CHR$(0);:CLOSE 2 : CLOSE 3:END
900 GOSUB 1000:Y$=X$:GOSUB 1000:NE=256*ASC(X$)+ASC(Y$):RETURN
1000 GET#2,X$:IF X$="" THEN X$=CHR$(0)
1010 RETURN
1100 Y$=CHR$(ME/256):X$=CHR$(ME AND 255):PRINT#3,Y$;X$;:RETURN
1200 FOR I=1 TO DB:GOSUB 1000:PRINT#3,X$;:NEXT I:RETURN
2000 GOSUB 900:IF NE=0 THEN RETURN
2010 DB=NE-MUT-2:MUT=NE:ME=MUT+EL
2020 GOSUB 1100:GOSUB 1200:GOTO 2000

```

5.5 Szekvenciális file-ok használata

A bevezetőben már vázoltuk, mik is azok a szekvenciális file-ok. Ebben a paragrafusban használatukról lesz szó.

A file megnyitása Mielőtt a file-t használnánk, a megfelelő OPEN utasítás segítségével meg kell nyitnunk. Ennek alakja:

```
OPEN lf, <egység>, <csatorna>, "<név>,<type>,<mód>"
```

lf a file logikai száma, ami alatt a BASIC interpreter nyilván-
tartásba veszi. Az <egység> értéke általában 8, de amennyiben
a lemezegységet átprogramoztuk, lehet ettől eltérő is. A
<csatorna> a lemezegység lehetséges 2-14 sorszámú adatcsator-
nái közül jelöl ki egyet. A <név> a megnyitandó file neve,
<type> pedig a típusa. <type> megengedett értékei a következők:

<u>type</u>	<u>jelentés</u>
PRG	program file
SEQ	szekvenciális file
USR	felhasználói file /5.8-ban/
REL	relativ file /5.8-ban/

<mód> lehetséges értéke READ vagy WRITE. A fenti szavak első
betűjükkal rövidíthetők.

Következő példánkban írásra megnyitunk egy szekvenciális file-t
és 25 egész számot írunk bele:

```
10 OPEN 15,8,15:PRINT#15,"S:PROBA"
20 OPEN 1,8,2,"PROBA,SEQ,WRITE"
30 FOR J=1 TO 25
40 X$="SZAM "+STR$(J)
50 PRINT#1,X$;CHR$(13);
60 S=ST
70 INPUT#15,E1,E2$,E3,E4
80 PRINT X$E1","E2$","E3","E4","S
90 NEXT J
100 CLOSE 1: CLOSE15
```


Az első sor törli az esetleg meglevő PROBA nevű file-t, majd a következő sor megnyitja írásra. Az 50. sorban történik az X\$ értékének, majd rögtön azt követően egy 'rekord vége' jelnek a kiírása a file-ba. Az X\$ így az INPUT utasítással majd visszaolvasható lesz. A 100. sorban a CLOSE 1 utasítás lezárja a PROBA-t.

Ugyanezt a file-t a következő programrésszel olvashatjuk vissza:

```
110 OPEN 15,8,15:OPEN 1,8,2,"PROBA,SEQ,READ":PRINT"7"
120 FOR J=1 TO 25
130 INPUT#1,X$
140 S=ST
150 INPUT#15,E1,E2$,E3,4
160 PRINT X$E1","E2$","E3","E4","S
170 NEXT J
180 CLOSE 1 :CLOSE 15:END
```

Ugyanezt a feladatot a GET# utasítás segítségével is megoldhatjuk:

```
110 OPEN 15,8,15:OPEN 1,8,2,"PROBA,SEQ,READ":PRINT"7"
120 X$=""
130 GET#1,A$:S=ST
140 IF A$="" THEN A$=CHR$(0)
150 IF A$<>CHR$(13) THEN X$=X$+A$:GOTO 130
160 INPUT#15,E1,E2$,E3,E4
170 PRINT X$E1","E2$","E3","E4","S
180 IF S=0 THEN 120
190 CLOSE 1 : CLOSE 15 : END
```

Az X\$ előállítását 130-160. sorok végzik. Az X\$-hoz egészen addig adjuk hozzá az utoljára beolvasott A\$ karaktert, míg egy CHR\$(13) jelet nem olvasunk /150. sor/. Ezután kerül sor az X\$ illetve a hibaüzenet kiírására. A program előnye, hogy ismeretlen számú rekordot is feldolgozhatunk a segítségével. /Valódi haszon persze akkor van, ha nem ismerjük a rekord-strukturát./

5.6 Véletlen file-ok használata

A bevezetőben már említettük, hogy lehetőség nyílik az egyes blokkok közvetlen írására, illetve olvasására. Ahhoz, hogy a DOS ebben az üzemmódban dolgozzon egy **#** nevű pseudo-file-t kell megnyitnunk, ami valójában a lemezegység valamelyik puffert jelent. /Ezenkívül mindenképp szükségünk van a 15. csatorna megnyitására is/:

```
OPEN lf, <lemezegység>, <csatorna>, "# [<sorszám>]"
```

lf a file logika file-száma, amire az I/O utasításokban hivatkoznunk kell, <lemezegység> a lemez száma /ez általában 8/; <csatorna> azt az adatcsatornát jelenti, amin keresztül az adatátvitel zajlik. Értékének a 2-14 tartományba kell esnie. Az opcionális <sorszám> a puffer sorszámát adja meg, ha elmarad a DOS maga választja meg, melyik puffert használja. Például

```
OPEN 15,8,15 : OPEN 1,8,2,"# "
```

utasítja a DOS-t a közvetlen elérésű üzemmód használatára. A lemez és a puffer közti adatátvitelt a 15. csatornán DOS-parancsokkal szabályozhatjuk. A puffert az **INPUT#** és a **GET#** utasításokkal, mint 1-es file-t olvashatjuk, illetve a **PRINT#** utasítással írhatunk bele.

A következőkben összefoglaljuk a közvetlen elérésű üzemmód DOS-parancsait.

B-R

BLOCK-READ utasítás

A DOS-parancs lehetővé teszi egy tetszőleges blokk olvasását a lemezeről.

Szintaxis: PRINT#hf, "BLOCK-READ: "; <csatorna>; <meghajtó>; <sáv>;
<szektor> vagy

PRINT#hf, "BLOCK-READ:<sztring>"

Az utasítás fenti alakjában a hf a 15. csatornának megfelelő logikai file-szám /ami általában 15/. <csatorna> a # pszeudo-file megnyitásában szereplő csatornaszám. <meghajtó> jelen esetben mindig 0. /Olyan lemezegység esetén, amelyik két lemez meghajtót is tartalmaz, <meghajtó> értéke 0 vagy 1 lehet./ A <sáv> és <szektor> annak a blokknak a paraméterei, amit a pufferbe szeretnénk beolvasni. Az utasítás második formájában a <sztring> négy karakterének ASCII kódja felel meg a fenti négy értéknek. A "BLOCK-READ:" szövegrész az utasításban a "B-R:" és "U1:" szöveggel helyettesíthető. /Lásd a USER utasítást./

Következő példánk a lemez megadott blokkját olvassa be a C-64 memóriájába és írja ki a képernyőre:

```
10 OPEN 15,8,15:OPEN 1,8,2,"#"
20 INPUT "SAV,SZEKTOR";S,SZ
30 PRINT#15,"B-R: ";2;0;S;SZ
40 GET#1,X$:IF ST=64 GOTO 20
50 PRINT X$;:GOTO 40
```

A szekvenciális file-okról szóló részben említettük, hogy a szekvenciális file-ok blokkjai első 2 byte-ja a következő blokk paramétereit adja meg. A következő program lehetővé teszi az egymáshoz láncolt blokkok követését:

```
10 OPEN 15,8,15:OPEN 1,8,2,"#"
20 INPUT "SAV,SZEKTOR";S,SZ
30 PRINT#15,"B-R: ";2;0;S;SZ
35 PRINT#15,"B-P: ";2;0
40 GET#1,X$:IF X$="" THEN X$=CHR$(0)
50 GET#1,Y$:IF Y$="" THEN Y$=CHR$(0)
55 S=ASC(X$):SZ=ASC(Y$)
60 PRINT "SAV=";S,"SZEKTOR=";SZ
65 GET A$:IF A$="" THEN GOTO 65
70 IF S=0 THEN CLOSE1:CLOSE15:END
75 GOTO 30
```

B-W

BLOCK-WRITE utasítás

Ez a DOS-parancs lehetővé teszi egy tetszőleges blokk írását.

Szintaxis: PRINT hf, "BLOCK-WRITE: "; <csatorna>; <meghajtó>; <sáv>;

<szektor> vagy

PRINT hf, "BLOCK-WRITE:" +<sztring>

Az utasítás fenti alakjában a hf a 15. csatornának megfelelő logikai file szám /ami általában ugyancsak 15/. A csatorna a # pseudo-file megnyitásában szereplő csatornaszám. A <meghajtó> jelen esetben mindig \emptyset . /Olyan lemezegység esetében, amelyik két lemezmeghajtót is tartalmaz, a <meghajtó> értéke \emptyset vagy 1 lehet./ A <sáv> és <szektor> annak a blokknak a paraméterei, ahová a puffer tartalmát ki akarjuk írni. Az utasítás második formájában a <sztring> négy karakterének ASCII kódja felel meg a fenti négy paraméternek. A "BLOCK-WRITE:" szövegrész az utasításban a "B-R:" és "U2:" szöveggel is helyettesíthető. /Lásd a USER utasítást./

Első példánk az 1. sáv valamennyi /0-20 sorszámú/ szektorába beleírja ugyanazt az üzenetet. A szektorok sorrendjének változtatásával ki lehet próbálni, melyik esetben a leggyorsabb a DOS.

```
10 OPEN 2,8,2,"#":OPEN 15,8,15
20 DATA 0,1,2,3,4,5,6,7,8,9,10
30 DATA 11,12,13,14,15,16,17,18,19,20
40 INPUT "ÜZENET";M$
45 FOR I=1 TO 21:READ SZ
50 PRINT#15,"B-R: ";2;0;1;SZ
60 PRINT#15,"B-P: ";2;0
70 PRINT#2,M$;
80 PRINT#15,"B-W: ";2;0;1;SZ
85 NEXT I
90 CLOSE2:CLOSE15:END
```

Második példánk egy adott blokkot részben /adott pozíciótól kezdődően/ módosít:

```

10 OPEN 1,8,2,"*":OPEN 15,8,15
20 INPUT "SAV,SZEKTOR";S,SZ
30 INPUT "KEZDOPOZICIO";P
40 INPUT "UZENET";M$
50 PRINT#15,"B-R:";2;0;S;SZ
60 PRINT#15,"B-P:";2;P
70 PRINT#1,M$;
80 PRINT#15,"B-W:";2;0;S;SZ
90 CLOSE1:CLOSE15:END

```

B-E

BLOCK-EXECUTE utasítás

Az utasítás hatása hasonló egy program betöltéséhez majd futtatásához. A lemeznek az utasításban specifikált blokkja betöltődik a pufferba és a program végrehajtása a puffer elején folytatódik. Amikor a gépi kódú program egy RTS-hez ér, az a BASIC programba való visszatérést eredményezi. Ritkán használjuk, mert ehhez a DOS ROM részletes ismeretére van szükség.

Szintaxis: PRINT#hf, "BLOCK-EXECUTE:"; <csatorna>; <meghajtó>; <sáv>;
<szektor> vagy
PRINT#hf, "BLOCK-EXECUTE:" + <sztring>

Az egyes paraméterek jelentése megegyezik a B-W és B-R parancsoknál leírtakkal. A "B-E:" rövidítés használata megengedett.

B-A

BLOCK-ALLOCATE utasítás

Az utasításban specifikált blokknak a BAM-ban megfelelő bit alacsony lesz, jelezve a DOS-nak, hogy azt más célokra már

nem használhatja. Amennyiben a blokk már foglalt, 65, NO BLOCK, S, SZ hibajelzést kapunk, ahol S, SZ a következő, még nem foglalt blokk sáv- és szektorszámát adja meg.

Szintaxis: PRINT#hf, "BLOCK-ALLOCATE:"; ~~<csatorna>~~ <meghajtó>;
<sáv>; <szektor> vagy
PRINT#hf, "BLOCK-ALLOCATE:" + <sztring>

A B-A parancs paramétereinek jelentése megegyezik a B-F parancsban leírtakkal. A "B-A:" rövidítés használata megengedett.

Példánk egy szubrutint mutat be, amelyik a következő üres /nem foglalt/ blokk sáv- és szektorszámával tér vissza. A 65-ös hiba figyelésén kívül még arról is gondoskodni kell, hogy a blokk - néhány kivételtől eltekintve - nem lehet a katalógus része. A program az S, SZ értékpárban adja vissza a következő szabad blokk paramétereit. E értéke a hibakódot tartalmazza. Ez 0, ha sikerült szabad blokkot találni, különben a felmerült DOS-hiba kódját tartalmazza:

```
1000 PRINT#15,"B-A:";0;S;SZ
1010 INPUT#15,E,E$,ES,EZ
1020 IF E=0 THEN RETURN
1030 IF E<>65 THEN RETURN
1040 S=ES:SZ=EZ:IF S=18 THEN S=19
1050 GOTO 1000
```

B-F

BLOCK-FREE utasítás

A DOS-parancs a B-A parancs ellentettje. A B-F parancsban specifikált blokknak megfelelő bit magas lesz, jelezve, hogy a továbbiakban a DOS azt más célokra felhasználhatja.

Szintaxis: PRINT#hf, "BLOCK-FREE:"; <meghajtó>; <sáv>; <szektor>

Az egyes paraméterek jelentése megegyezik a B-R illetve a B-W utasításban leírtakkal. A csatorna megadására értelem-szerűen nincs szükség. A "B-F:" rövidítés megengedett. A

```
PRINT #15, "B-F: "; 0; 1; SZ
```

az 1. sáv SZ-ik szektorát a DOS rendelkezésére bocsátja.

B-P

BUFFER-POINTER utasítás

A # pszeudo-file-hoz egy mutató is tartozik, amelyik a lehetséges /1-255-ig számozott/ byte valamelyikére mutat. Az író/olvasó utasítások végrehajtása ettől a byte-tól kezdődik. A mutatót a B-P utasítás segítségével tetszőleges helyre pozícionálhatjuk. A mutató utolsó értéke, mint a blokk 0. byte-ja a lemezre íródik.

Szintaxis: PRINT #hf, "BUFFER-POINTER: "; <csatorna>; <mutató>

A hf és a <csatorna> jelentése ugyanaz, mint a B-R utasításban; <mutató> a # pszeudo-file pufferének pointerét állítja át. A "B-P:" rövidítés itt is megengedett.

A B-P parancsot elsősorban a B-W és B-R utasításokkal együtt használjuk. /B-W példájában már szerepelt./

M-R

MEMORY-READ utasítás

Az utasítás lehetővé teszi a lemezegység memóriájának byte-onkénti olvasását. /Akár a RAM-ból, akár a ROM-ból olvashatunk./ Ez például lehetővé teszi a DOS működésének tanulmányozását, a 0. lap megismerését stb. Minden egyes byte-re egy új utasítást kell kiadni.

Szintaxis: PRINT*hf, "M-R: "; CHR\$(L); CHR\$(H), ahol

hf a lemezegység 15. csatornájához tartozó logikai file szám; L és H a memória címének alsó illetve felső byte-ja. Az utasítást követően a memória címén levő byte a GET*hf, A\$ utasítás segítségével olvasható.

M-E

MEMORY-EXECUTE utasítás

Az utasítás lehetővé teszi a lemezegység tetszőleges címén kezdődő /gépi kódú/ program végrehajtását. Ez lehet egy ROM alprogram, de lehet a RAM-ban az M-W parancs segítségével megírt program is. A szintaxis megegyezik az M-R utasítás szintaxisával.

Szintaxis: PRINT*hf, "M-E: "; CHR\$(L); CHR\$(H), ahol

L és H a kezdőcím alsó illetve felső byte-ja.

M-W

MEMORY-WRITE utasítás

A parancs lehetővé teszi - egyidőben 34 byte - beírását a DOS memóriájába. Ezeket azután az M-E parancs használhatja.

Szintaxis: PRINT*hf, "M-W: "; CHR\$(L); CHR\$(H); <karakterek száma>;
<byte-sorozat>

hf, L, H jelentése ugyanaz, mint a M-R utasításban. Ezt követi a <karakterek száma> paraméter, amelyik definiálja, hány byte-ból áll a DOS memóriájába írandó sorozat. Ezt követik a sorozat byte-jai CHR\$(B) alakban. Például

```
PRINT# 15, "M-W: "; CHR$(0); CHR$(5); 1; CHR$(96)
```

a \$0500 címre egyetlen byte-ot /96/ helyez el. Ez egy RTS utasításnak felel meg.

USER

Ez a DOS-parancs lehetővé teszi a DOS memóriájában tárolt bizonyos címekre való ugrást. Ezek általában további JMP utasításokat használnak, amelyek lehetővé teszik egy tetszőleges DOS rutin elérését.

Szintaxis: PRINT # hf, "<USER utasítás>"

hf a 15. csatorna megnyitásában használt logikai file szám /általában maga is 15/. Az utasításhoz néha további byte-okat is el kell küldeni. A <USER utasítás> lehetséges értékeit és azok jelentését a következő táblázat foglalja össze:

<u>USER utasítás</u>	<u>Jelentés</u>
U1 vagy UA	B-R a puffer-pointer felhasználása nélkül
U2 vagy UB	B-W a puffer-pointer felhasználása nélkül
U3 vagy UC	JMP \$0500
U4 vagy UD	JMP \$0503
U5 vagy UE	JMP \$0506
U6 vagy UF	JMP \$0509
U7 vagy UG	JMP \$050C
U8 vagy UH	JMP \$050F
U9 vagy UI	JMP \$FFFA /=NMI vektor/
U; vagy UJ	RESET vektor
UI+	C-64 sebességének kiválasztása
UI-	VIC 20 sebességének kiválasztása

Külön szólnunk az U1 és U2 hatásáról. A B-R utasítás az adott blokkot csak a blokk első byte-jaként tárolt puffer-mutató értékéig olvassa. Ha az utasítás U1 alakját használjuk, akkor mint a 256 byte a pufferbe kerül. A B-W utasítás a puffer-pointer értékét és mind a 255 adat-byte-ot az adott blokkba írja. Az utasítás U2 alakja a puffert a lemezen levő mutató értékéig másolja csak vissza.

5.7 Gépi kódu programok

A bemutatott, lemezegységet használó gépi kódu programok, programrészek elsősorban a KERNAL rutinokat használják, így megértésükhöz a 9. és a 6. fejezet mélyebb ismeretére van szükség. A gépi kódu programrészek megírásához a KERNAL rutinokon kívül nagy segítséget nyújtanak a következő, \emptyset . lapon levő címek:

<u>Cím</u>	<u>Leírás</u>
183 /\$00B7/	Aktuális file név /parancs/ hossza
184 /\$00B8/	Aktuális logikai file szám
185 /\$00B9/	Aktuális csatornaszám
186 /\$00BA/	Aktuális egységyszám
187-188 /\$00BB-00BC/	Aktuális file név /parancs/ elejére mutat.
/i/	

033C	10	*= \$033C
033C R2 02	20	LDX #\$02
033E 20 C6 FF	30	JSR \$FFC6
0341 A0 00	40	LDY #\$00
0343 20 CF FF	50 L1	JSR \$FFCF
0346 99 00 04	60	STA \$0400,Y
0349 A9 0E	70	LDA #\$0E
034B 99 00 D8	80	STA \$D800,Y
034E C8	90	INY
034F F0 04	100	BEQ OUT
0351 A5 90	110	LDA \$90
0353 F0 EE	120	BEQ L1
0355 4C CC FF	130 OUT	JMP \$FFCC

Programunk egy már BASIC vagy gépi kódu rutinnal megnyitott szekvenciális file adatait olvassa és írja ki a képernyő tetejére. A képernyő és ASCII kódok eltérnek, ezért a 'rekordvége' jel /CHR\$(13)/ mint egy M betű fog megjelenni. A programot az

```
OPEN 2,8,2, "FILE,SEQ,READ" : SYS 828
```

paranccsal hajthatjuk végre.

/ii/ Egy parancs sztring elküldése A legtöbb esetben a parancs az input pufferben található, ez azonban nem törvényszerű. A parancs elküldéséhez a következőket kell végrehajtánunk:

- /i/ \$08 betöltése \$BA-ba /egységszám = 8/
- /ii/ \$6F betöltése \$B9-be /csatornaszám = 15 + TALK/
- /iii/ 'LISTEN' üzenet elküldése
- /iv/ Az IEEE parancs / \$6F/ elküldése
- /v/ A parancs sztring byte-onként való elküldése
- /vi/ 'UNLISTEN' üzenet elküldése.

A gépi kódu rutint a SYS 828,"<parancs> típusu aktivizálásra terveztük, első használata előtt az OPEN 15,8,15 utasítást nem kell kiadni. A program elején a használt KERNAL rutinok kezdőcimeit felsoroltuk.

```

1000          10          ; PARANCs ELKULDESE
1000          20          ;
1000          30          ; HIVASA:
1000          40          ;          SYS 828,"<PARANCs>
1000          50          ;
FFA8          60 CIOUT    = $FFA8
FF93          70 SECOND  = $FF93
0073          80 CHRGET  = $73
FFB1          90 LISTEN  = $FFB1
00BA         100 FA      = $BA
00B9         110 SA      = $B9
FFAE         120 UNLSN   = $FFAE
033C         130        * = $033C
033C 20 73 00 140      JSR CHRGET    ; A (,) ATLEPESE
033F A9 08    150      LDA #$08
0341 85 BA    160      STA FA        ; EGYSEGSZAM = 8
0343 A9 6F    170      LDA #$6F
0345 85 B9    180      STA SA        ; CSATORNA = 15
0347 A5 BA    190      LDA FA
0349 20 B1 FF 200      JSR LISTEN
034C A5 B9    210      LDA SA
034E 20 93 FF 220      JSR SECOND
0351 20 73 00 230  LOOP JSR CHRGET
0354 C9 00    240      CMP #$00
0356 F0 06    250      BEQ OUT
0358 20 A8 FF 260      JSR CIOUT
035B 4C 51 03 270      JMP LOOP
035E 20 AE FF 280  OUT JSR UNLSN
0361 60      290      RTS

```

Például SYS 828,"I inicializálja a lemezegységet,
SYS 828,"S:PROG× törli az összes PROG-gal kezdődő file-t stb.

/iii/

033C		10		*=033C	
033C	A9 08	20		LDA #\$08	; EGYSEG SZAM = 8
033E	85 BA	30		STA \$BA	
0340	20 B4 FF	40		JSR \$FFB4	
0343	A9 6F	50		LDA #\$6F	; CSATORNA = 15
0345	20 96 FF	60		JSR \$FF96	; HIBA CSATORNA
0348		65		;	KOVETKEZO BYTE
0348	20 A5 FF	70	LOOP	JSR \$FFA5	
034B	C9 0D	80		CMP #\$0D	
034D	F0 05	90		BEQ OUT	
034F	20 D2 FF	100		JSR \$FFD2	
0352	D0 F4	110		BNE LOOP	
0354	20 D2 FF	120	OUT	JSR \$FFD2	
0357	20 AB FF	130		JSR \$FFAB	
035A	60	140		RTS	

Utolsó példánk a hiba csatornát olvassa és tartalmát kiírja a képernyő tetejére. A program ugyancsak a SYS 828 paranccsal hajtatható végre.

5.8 Relativ file-ok használata

A DOS két olyan file-típus használatát is biztosítja, amiről eddig még nem szóltunk. Ezek a katalógusban a `USR` és `REL` típusjelzésnek megfelelő felhasználói és relativ file-ok. A felhasználói file-okat a DOS 1.X verzióju változatai használták és többé-kevésbé ugyanazt a feladatot látta el, mint a DOS 2.X verzióju változataiban a relativ file-ok. A DOS 2.6 a felhasználói file-okat gyakorlatilag nem különbözteti meg a szekvenciális file-októl. Ebben a paragrafusban a relativ file-ok lehetőségeiről lesz szó.

A relativ file-ok, hasonlóan a szekvenciális file-okhoz, rekordokból állnak, de ezek a rekordok ugyanolyan hosszúak. /Éppen ezért nincs szükség a rekord végét jelölő speciális karakterek használatára./ Szemben azonban a szekvenciális file-okkal, ezekre a rekordokra sorszámuk alapján lehet hivatkozni. Ahhoz, hogy a DOS gyorsan megtaláljon egy-egy adott rekordot, külön blokkokra van szükség, amelyek megadják, hogy egy-egy rekord melyik blokkon található. Erre a célra a DOS összesen hat blokkot foglal le. Ezért aztán egy-egy relativ file-ban maximum 720 rekord lehet.

Relativ file-ok megnyitása/lezárása Egy még nem létező relativ file-t a következő utasítással nyithatunk meg:

```
OPEN lf, <egységszám>, <csatorna>, "<név>,L," + CHR$( <hossz> )
```

`lf` a logikai file szám, `<név>` a relativ file neve, `<csatorna>` a programban még eddig nem használt, 2-14 adatcsatornák bármelyike lehet. Az "L" betű a relativ file-t jelenti. `<egységszám>` a lemezes egység száma, általában 8. `<hossz>` a relativ file rekord hosszát adja meg és legfeljebb 254 lehet. A file egész 'élete' alatt ez lesz a rekordhossz. Az utasítás fenti formájában hiába használjuk a `@:<név>` alakot, a file nem fog törlődni. A relativ file-ok csak a `SCRATCH` paranccsal törölhetők.

Már meglevő relatív file-t az

```
OPEN lf, <egységszám>, <csatorna>, "<név>"
```

utasítással lehet megnyitni. A READ és WRITE megjelölésre nincs külön szükség; lévén, hogy a relatív file-ba írni, olvasni egyszerre lehet.

A megnyitott file-t a

```
CLOSE lf
```

utasítással zárhatjuk le.

A rekordszám beállítása Relatív file-okat ugyanugy írhatunk, olvashatunk mint szekvenciális file-okat, azzal a lényeges különbséggel, hogy megadhatjuk hányadik rekord hányadik pozíciójától kezdődjék az I/O művelet. Erre szolgál a P DOS-parancs, melynek alakja:

```
PRINT#hf, "P" + CHR$(<csatorna>) + CHR$(L) + CHR$(H) + CHR$(P)
```

hf a 15. csatorna megnyitásakor használt logikai file szám, <csatorna>a relatív file megnyitásában használt csatornaszám /secondary adress/. L és H a rekord sorszámának alsó és felső byte-ja. A valódi sorszám tehát $256 \times H + L$. Végül P a rekordon belül a kezdő byte-pozíció.

Azt, hogy egy relatív file-nak hány rekordja van, a file 'élete' során a P parancsban kiadott legnagyobb rekordszám dönti el. Ha a file hosszánál nagyobb rekordszámmra hivatkozik a P parancs, a DOS helyet foglal a lemezen a további rekordoknak. A még nem használt rekordok első byte-ja \$FF, a többi 0. Ha egy rekordot nem írunk tele, a maradék rész 0-kal lesz tele.

Relatív file-ok írása/olvasása A GET#, INPUT# illetve PRINT# utasításokat használhatjuk a relatív file-ba való írásra és olvasásra. Az I/O művelet a legutolsó P DOS-parancsban

megadott pozíciótól kezd a file-t feldolgozni. GET az azon a pozíción levő karaktert adja vissza, INPUT* a legközelebbi CHR\$(13)-ig olvassa a file-t /akár a rekord végén túl is!/, PRINT# az adott pozíciótól kezdve ír és ugyancsak átlépi a rekordhatárt. Kivétel természetesen a file utolsó rekordja, amelyen ha túl olvasunk vagy tulirunk, hibajelzést kapunk /ST=64/. Mivel azonban a rekord vége általában logikai határ is, célszerű elkerülni az olyan I/O műveletet, amelyik átlépi a rekord határát.

Példánkban egy számokat tartalmazó file-t nyitunk meg, és be-
leirunk 30 számot. A következő programrész segítségével tet-
szőleges számot visszaolvashatunk a lemezről és ha akarjuk,
módosíthatjuk. Egy-egy számot a relatív file egy rekordja tar-
talmaz.

```

10 OPEN 15,8,15:PRINT#15,"S:PROBA"
20 OPEN 1,8,2,"PROBA,L,"+CHR$(14)
30 FOR I=0 TO 29:SZAM=I:REM INPUT"KOVETKEZO SZAM=";SZAM
40 PRINT#15,"P"+CHR$(2)+CHR$(I)+CHR$(0)+CHR$(0)
45 INPUT#15,E1,E2$,E3,E4:IF(E1<20)OR E1=50 THEN GOTO 50
46 PRINT E1,E2$,E3,E4:END
50 PRINT#1,SZAM;
55 INPUT#15,E1,E2$,E3,E4:IF(E1<20)OR E1=50 THEN GOTO 60
56 PRINT E1,E2$,E3,E4:END
60 NEXT:CLOSE1:CLOSE15
70 OPEN 15,8,15:OPEN 1,8,2,"PROBA"
80 INPUT"MELYIK SZAM KELL";I
90 I=INT(I):IF (I<1) OR (I>30) GOTO 80
100 PRINT#15,"P"+CHR$(2)+CHR$(I)+CHR$(0)+CHR$(0)
110 INPUT#1,SZAM
120 PRINT SZAM
130 PRINT"AKARJA MODOSITANI (I/N) ?"
140 GET A$:IF A$="" THEN 140
150 IF A$="N" THEN GOTO 190
160 INPUT"UJ SZAM ERTEKE=";SZAM
170 PRINT#15,"P"+CHR$(2)+CHR$(I)+CHR$(0)+CHR$(0)
180 PRINT#1,SZAM;
190 PRINT "ELEG VOLT (I/N) ?"
210 GET A$: IF A$="" THEN 210
220 IF A$="N" THEN GOTO 80
230 CLOSE1:CLOSE15:END

```

Utószó a második kiadáshoz

A Commodore 64 BASIC Felhasználói Kézikönyv első kiadása néhány hét alatt elfogyott. Ez a siker - gondolom - nem elsősorban a könyv érdemeinek, hanem a C-64 mikroszámítógép iránti érdeklődésnek köszönhető. Hazánkban jelenleg közel háromezer C-64 típusu gép található, és számuk - egyes szakértők becslése szerint - az év végére eléri a négyezert. Ilyen kereslet mellett az első kiadás ötezer példányszáma valóban nem túl sok.

Számomra úgy tűnik, hogy a C-64-hez nyújtott - hazai !/-hardver és szoftver szolgáltatások színvonala egyedülállóan magas a többi mikroszámítógéphez képest. A szoftver, hardver fejlesztésében a legnagyobb rendszerházaktól az "egyszemélyes" GMK-ig szinte mindenki résztvesz.

A KSH Számítástechnika-alkalmazási Főosztálya, illetve az LSI Alkalmazástechnikai Tanácsadó Szolgálat által az 1984. évi tavaszi BNV-re megjelentetett mikroszámítógépes katalógusok együtt közel 200 szoftver, illetve hardver terméket kínálnak a C-64-re. /Az egyik az SKV, a másik az Akadémia Kiadó könyvesboltjában kapható/. A termékek skálája igen széles. A C-64 összekapcsolása ESzR, MSzR és IBM gépekkel /az RS232 csatorna segítségével/ megtörtént. A C64-hez a legkülönfélébb terminál emulátorokat kínálják. A perifériális eszközök kínálata is bőséges a magyar ABC-s karakterkészletű nyomtatótól a szabványos numerikus billentyűzetig. Az alapszoftver területén szövegszerkesztőket, fordítóprogramokat, file-kezelő rendszereket, egyéb programnyelveket /pl. FORTH/ vásárolhatunk /borsos áron persze/. A SZÁMALK OSAK - más néven ugyan - de forgalmazza a könyvben is szereplő HELP+-t és magyarosított alapszavakkal a SIMONS' BASIC-et. Legbősegebb a választék természetesen az alkalmazói szoftverek területén. Ezek színvonala igen

eltérő, ezért megvásárlásuk előtt célszerű alaposan kipróbálni őket.

Végül az ut/ols/ó szó jogán néhány további kiegészítést teszünk és végül felsoroljuk az első kiadásban talált értelmetlen zavaró hibákat.

A perifériák bekapcsolása

Az 1.4 oldal 6-os pontjában irtakkal ellentétben a perifériákat nem mindig lehet a C-64 bekapcsolása előtt bekapcsolni. Ez különösen több lemezegység használatánál okoz problémát. Ilyenkor először a C-64-et kell bekapcsolni, majd egymás után a lemezegységeket. Ha erre szükség van /mert a lemezegységeket a hardver segítségével nem számoztuk át/, menet közben az egyes egységeket át is kell számozni.

KERNAL rutinok

A 9.4 pontban ismertettük a KERNAL rutinokat. Ezek azok a rutinok, melyek belépési pontjait a \$FF81 címen kezdődő táblázat tartalmazza. Néhány rutin a 3. lapon levő RAM-címeket használja belépési pontként, így ezeket tetszés szerint átírhatjuk. Így például - a BASIC bolygatása nélkül - lehetőség nyílik egy nem CBM kompatibilis nyomtató illesztéséhez, mondjuk 5-ös egység számmal. Ezek a rutinok a következők:

/\$31A/ = OPEN
/\$31C/ = CLOSE
/\$31E/ = CHKIN
/\$320/ = CHKOUT
/\$322/ = CLRCHN
/\$324/ = CHRIN
/\$326/ = CHROUT
/\$328/ = STOP
/\$32A/ = GETIN
/\$32C/ = CLALL
/\$330/ = LOAD
/\$332/ = SAVE

További két KERNAL rutint tartalmaz a tábla.

IOBASE

Belépési pont: \$FFF3 (65523)

Használt regiszterek: X,Y

Előkészítő rutinok: -

A rutin az I/O eszközöket használó memóriarész kezdőcímével tér vissza az X, Y regiszterekben. X tartalmazza a cím alsó, Y pedig a felső byte-ját.

SETTMO

Belépési pont: \$FFA2 (65442)

Használt regiszterek: A

Előkészítő rutinok: -

A rutin abban az esetben használható, ha a bővítő egység kimenetébe egy IEEE kártyát helyeztünk. Ebben az esetben a rutin segítségével az IEEE busz timeout jelzőjét állíthatjuk be. A C-64 64 ezredmásodpercig vár a válaszra, és ha az nem érkezik meg, hibajelzést generál és abbahagyja a /handshake/ szinkronizálást. Ha a rutin meghívásához az A regiszter 7. bitje magas, a timeout jelző beállítódik. Ha a bit alacsony, a jelző kikapcsolódik.

A KERNAL rutinok az átvitel /C/ bitet használják hibajelzésre. Ha a bit alacsony (C=0), a rutin hiba nélkül lefutott. Ha magas, a rutin nem hajtott végre szabályszerűen végre és a hiba kódja ilyenkor az akkumulátorba (A regiszter) kerül.

Ennek értékei a következők lehetnek:

Érték	Jelentés
0	A rutin futását a STOP lenyomásával megállítottuk
1	Túl sok nyitott file
2	A file-t már megnyitottuk
3	A file nincs megnyitva
4	A file-t nem találja a rendszer
5	Az egység nincs jelen
6	A file nem egy input file
7	A file nem egy output file
8	A file neve hiányzik
9	Nem megengedett egységszám
240	Az RS232 megnyitása/lezárása módosította a BASIC munkaterület vége mutatót

A KERNAL rutinok a soros busz használatát két szinten teszik lehetővé. Egyrészt a TALK, LISTEN stb. rutinok használatával, erre mutattunk példát az 5.7 pontban, másrészt a CHKOUT, CHKIN, CHROUT, CHRIN rutinok segítségével. Erre adunk egy egyszerű példát a következő oldalon.

A program karaktereket olvas be a billentyűzetről, ezt kiírja a képernyőre, illetve egy PROBA nevű szekvenciális file-ba. Az adatbevitelt a billentyű lenyomásával fejezhetjük be. A file beolvasását, illetve a képernyőre való kiírását a program második része végzi el.

```

100 CLALL = $FFE7 ; LEZARJA AZ OSSZES FILE-T
102 CHROUT = $FFD2 ; EGY BYTE KIIRASA
104 CHKOUT = $FFC9 ; AZ OUTPUT CSATORNA MEGNYITASA
106 CHRIN = $FFCF ; EGY BYTE BEOLVASASA
108 CHKIN = $FFC6 ; AZ INPUT CSATORNA MEGNYITASA
110 CLOSE = $FFC3 ; FILE LEZARASA
112 OPEN = $FFC0 ; FILE MEGNYITASA
114 SETNAM = $FFBD ; PARANCS BEIRASA
116 SETLFS = $FFBA ; FILE PARAMETEREINEK BEIRASA
118 ;
120 *=828
122 ;
124 JMP BE ; BELEPESI PONT IRASHOZ
126 ;

```

```

128      JMP KI ; BELEPESI PONT OLVASASHOZ
130      ;
132 BE    LDA #"W ; IOBYTE IRAST JELENT
134      JSR NYIT ; PROBA MEGNYITASA
136 BEOLV  LDA #$03 ; KARAKTEREK BEOLVASASA
138      JSR CHKIN ; A BILLENTYUZET KIVALASZTASA
140      JSR CHRIN ; EGY BYTE BEOLVASASA
142      TAY
144      CMP #"E ; ELERTUK-E A VEGET?
146      BEQ VEGE1
148      LDX #$02 ; A PROBA FILE KIVALASZTASA
150      JSR CHKOUT
152      TYA
154      JSR CHROUT ; EGY BYTE KIIRASA
156      JMP BEOLV
158      ;
160 VEGE   LDA #$02 ; A FILE LEZARASA
162      JSR CLOSE
164      JMP CLALL
166      ;
168 VEGE1  LDX #$02 ; A VEGET JELZO E KIIRASA
170      JSR CHKOUT
172      LDA #"E
174      JSR CHROUT
176      JMP VEGE
178      ;
180 KI     LDA #"R ; AZ IOBYTE OLVASAST JELENT
182      JSR NYIT ; A PROBA MEGNYITASA
184      LDX #$00
186      JSR CHKOUT
188      LDX #$02 ; A PROBA FILE KIVALASZTASA
190      JSR CHKIN
192 KAP    JSR CHRIN ; EGY BYTE BEOLVASASA
194      CMP #"E
196      BEQ VEGE
198      JSR CHROUT
200      JMP KAP
202 NYIT   STA IOBYTE ; A PARANCS BEIRASA
204      LDA #$09
206      LDY CIM+1
208      LDX CIM
210      JSR SETNAM ; A PARANCS HELYREMASOLASA
212      LDA #$02
214      LDX #$08
216      LDY #$02
218      JSR SETLFS ; AZ OPEN PARAMETEREINEK ELOALLITASA
220      JSR OPEN ; A FILE MEGNYITASA
222      RTS
224      ;
226 CIM     .WORD SZOVEG ; A PARANCS KEZDOCIME
228 SZOVEG  .TEXT"PROBA,S," ; A PARANCS
230 IOBYTE  .BYTE "R" ; ES UTOLSO BYTE-JA
232      .END

```

Hibaigazítás

Az aláhuzott részek az eredeti szövegtől eltérő helyeket jelentik.

<u>Sor</u>	<u>Helyes szöveg</u>
7.a.12 /lásd az 1.18 oldalt/.
74.f. 7	<u>Szintaxis:</u> CMD <aritmetikai kifejezés> [<nyomtatási kép>]
96.f.11	tyüzzünk....
100.f. 1	10 GOSUB <u>5000</u>
100.f. 2	<u>20</u> <SZAMITASOK>
100.f. 5	GOSUB <u>10000</u>
103.f. 1 <u>Token:</u> \$8B (139) <u>Szintaxis:</u> NEXT [<változólista>]
129.a. 6 <u>7 = kis betűk/ nagy betűk</u> ...
146.f. 2	RIGHT\$ (X\$,N)= ...
147.a. 4	<u>Szintaxis:</u> RUN [<sorszám>]
174.f. 1	<u>RCOMP:</u> <utasítás> : ...
177.	Az oldal a 161. oldal elé való /!/
198.f. 4	<u>Szintaxis:</u> PRINT# hf, "BLOCK-ALLOCATE" ; <meghajtó> ; ...
199.f. 4	PRINT#15, "B-F:"; 0; 1; <u>SZ</u>
326.a. 7	156 DATA 33,135,125,31,165,125
357.f. 1-2	I/O kapu Az <u>1.</u> cím egy kétirányu adatkapu, amelynek adatirány /DDR, data-direction/ re- gisztere a <u>\$0000</u> címen található.
358.a.16	... Tekintsük például a LDA (<u>\$00,X</u>)
363.a. 8	\$90 144 BCC <módosítás> 2 <u>2*+</u>
371.a.17	\$C1 193 CMP (0.lap, x) 2 3
371.a.13	\$D1 209 CMP (0.lap), y 2 <u>5*</u>
375.a. 8	\$41 65 EOR (0.lap, x) 2 6
379.f. 4	\$A1 161 LDA (0.lap, x) 2 6
379.f. 8	\$B1 177 LDA (0.lap),y 2 <u>5*</u>

381.f.10 ... 0 → 7 6 5 4 3 2 1 0 → C

382.f.12 \$01 1 ORA (0.lap, x) 2 6

382.f.16 \$11 17 ORA (0.lap), y 2 5

387.a.17 \$E1 225 SBC (0.lap, x) 2 6

387.a.13 \$F1 241 SBC (0.lap), y 2 5

389.a.10 \$81 STA (0.lap, x) 2 6

389.a. 7 \$91 STA (0.lap), y 2 6

392.f. 8 ... amelyik a memória 1. címén ...

392.f. 9 ... adat irány regiszter a 0.címen ...

414.f. 3 ... végrehajt egy ~~R~~R 1,1..

418.f.14 ... a HELP+ ~~F~~F, ~~G~~G, ~~D~~D

429.f.15 A 70-100. ...

437.a. 8 JSR \$0079 ...

460.f. 7-9 A sorokat törölni kell /!/

469.a.13 A CIA ~~#~~2 chip B kapujának bitjei

469.a. 6 A CIA ~~#~~2 chip A kapujának 2. bitje

```

100 REM*****
110 REM* SZAMELMELET FELADATOK *
120 REM*****
121 :
122 REM MENU KIIRASA
130 PRINT "***** SZAMELMELET MENU *****"
140 PRINT"PRIMSZAM-E.....(1)"
150 PRINT"PRIMSZAM KERESES.....(2)"
160 PRINT"IKERPRIMSZAM KERESES..(3)"
170 PRINT"OSZTOK SZAMA.....(4)"
180 PRINT"VEGE.....(5)"
190 INPUT "MELYIKET VALASZTJA 1-5:";V
200 V=INT(V)
210 IF (V<1) OR (V>5) THEN GOTO 130
220 ON V GOTO 240,310,670,410,230
230 PRINT"":END
232 :
240 PRINT"***** PRIMSZAM-E ***"
250 INPUT"PRIM SZAM=";X:REM A SZAM BEOLVASASA
260 GOSUB 560: REM PRIMSZAMRUTIN
270 A$="PRIM."
280 IF J1=0 THEN A$="NEM "+A$
290 PRINT"MA(Z)";X;A$
300 GOTO 500 :

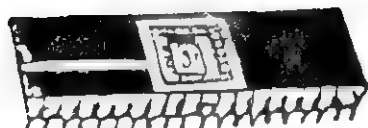
```

```

302 :
310 PRINT"***** PRIMSZAMKERESESES ***"
320 INPUT"HONNAN KERESSEM";N
330 X=N:A=2:GOSUB 520 : REM X AZ N UTANI ELSO
340 IF J1=0 THEN X=X+1: REM PARATLAN SZAM
350 GOSUB 560 : REM X PRIM-E?
360 IF J1=1 THEN 380
370 X=X+2:GOTO 350
380 PRINT"NA(Z)";N;"UTANI ELSO PRIM:"
390 PRINT:PRINTSPC(9);X
400 GOTO 500
410 PRINT"***** OSZTOK SZAMA ***"
420 INPUT"NA SZAM=";X.
430 S=0 : REM SZAMLALO AZ OSZTOK SZAMANAK
440 FOR A=1 TO X
450 GOSUB 520 : REM OSZTHATO-E A-VAL?
460 S=S+(1-J1): REM IGEN-->SZAMLALO NOVELESE
470 NEXT A
480 PRINT"NA(Z)";X;"-NEK";S;"OSZTOJA VAN."
490 GOTO 500
500 PRINT"NYOMJON MEG EGY BILLENTYUT!";
501 PRINT:
504 GET A$:IF A$="" THEN 500
510 GOTO 130
515 :
520 REM A J1 ERTEKE 0, HA A OSZTOJA X-NEK
530 J1=1
540 IFX=A*INT(X/A) THEN J1=0
550 RETURN
555 :
560 REM A J1 ERTEKE 1, HA X PRIMSZAM
590 IFX=1 THEN J1=0:RETURN: REM SPECIALIS ESETEK
600 IFX=2 THEN J1=1:RETURN
610 FOR A=2TOSQR(X)
620 GOSUB 520 : REM A OSZTOJA-E X-NEK?
630 IF J1=0 THEN RETURN : REM IGEN-->NEM PRIM
640 NEXT
650 RETURN : REM NEM VOLT OSZTOJA-->PRIM
660 :
670 PRINT"***** IKERPRIM KERESESE ***"
680 INPUT"HONNAN KERESSEM";N
690 X=N:A=2:GOSUB 520
700 IF J1=0 THEN X=X+1
710 GOSUB 560
720 IF J1=1 THEN X=X+2:GOTO 740
730 X=X+2:GOTO 710
740 GOSUB 560
750 IF J1=0 THEN GOTO 730
760 PRINT"NA(Z)";N;"-T KOVETO ELSO IKERPRIMEK:"
770 PRINT:PRINTSPC(8);X-2;SPC(8);X
780 GOTO 500

```

READY.



LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
LSI Application Information and Learning Centre
Budapest

OMIKK 1428 Bp Pf. 12.

Telefon 570-433/183 182, 185, 482, 270.
Telex OMIKK H 22-4944

TISZTELT CÍM!

COMMODORE-64 OKTATÓLEMEZ

A C-64 számítógép programozásának elsajátítását a lemez kétféleképpen segíti. Egyrészt programokat tartalmaz, amely a következő témákat foglalja magában:

- C-64 vezérlési struktúrák,
- I/O lehetőségek,
- VIC II chip használata,
- hanggenerálás,
- BASIC programozói fogások;

másrészt egy BASIC kiterjesztést tartalmaz a lemez, amely a következő lehetőségekkel bővíti a C-64 BASIC interpreterét:

- a C-64 utasításainak leírása,
- nyomkövetés,
- az I/O eszközök regisztereinek leírása,
- képernyő és ASCII kódok.

Az oktatólemez „önmagát bemutatja”, könyv nélkül is igen jól használható.

Ara: 4 000.- Ft

Megrendelhető: LSI ATSZ, Budapest, Pf. 12. 1428 (tel.: 573-374)

Országos Műszaki Információs Központ és Könyvtár
1088 Budapest, Múzeum u. 17. (tel.: 339-772).



LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
LSI Application Information and Learning Centre
Budapest

OMIKK 1428 Bp Pf 12.

Telefon: 570-433/183, 182, 185, 482, 276.
Telex OMIKK H 22-4944

TISZTELT CÍM!

COMMODORE-64 INFORMÁCIÓS KÁRTYA
(Quick Reference Card)

A keménykartonra nyomott néhány oldalas információs kártya (Quick Reference Card) összefoglalja a C-64 programozásához szükséges legfontosabb információkat. Ezek között a

- BASIC utasítások szintaxisa,
- M6510 utasításkészlete,
- I/O regiszterek,
- Memóriaszervezés,
- Képernyő és ASCII kódok.

Megjelenik: 1984 májusában.

Megrendelhető: LSI ATSZ, Budapest, Pf. 12. 1428 (tel.: 573-374) és

Országos Műszaki Információs Központ és Könyvtár,
Értékesítési Osztály
1088 Budapest, Múzeum u. 17. (tel.: 339-772)

M E G R E N D E L Ő L A P

Megrendeljük az LSI ATSZ kiadásában megjelenő

COMMODORE-64 INFORMÁCIÓS KÁRTYÁT

..... példányban.

Megrendelő neve:

Címe (irányítószám):

Megrendelés száma:

Ügyintéző:

Kelt:

.....
cégszerű másolás



TISZTELT CÍM!

COMMODORE-64 INFORMÁCIÓS KÁRTYA
(Quick Reference Card)

A keménykartonra nyomott néhány oldalas információs kártya (Quick Reference Card) összefoglalja a C-64 programozáshoz szükséges legfontosabb információkat. Ezek között a

- BASIC utasítások szintaxisa,
- M6510 utasításkészlete,
- I/O regiszterek,
- Memóriaszervezés,
- Képernyő és ASCII kódok.

Megjelenik: 1984 májusában.

Megrendelhető: LSI ATSZ, Budapest, Pf. 12. 1428 (tel.: 573-374) és

Országos Műszaki Információs Központ és Könyvtár,
Értékesítési Osztály
1088 Budapest, Múzeum u. 17. (tel.: 339-772)

MEGRENDELŐLAP

Megrendeljük az LSI ATSZ kiadásában megjelenő

COMMODORE-64 INFORMÁCIÓS KÁRTYÁT

..... példányban.

Megrendelő neve:

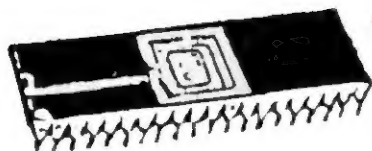
Címe (irányítószám):

Megrendelés száma:

Ügyintéző:

Kelt:

.....
cégszerű aláírás



LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT

LSI Application Information and Learning Centre

Levélcím: OMIKK – LSI ATSz 1428 Bp. Pf.: 12

Bp. 1103 Noszlopy u. 1.

Telefon: 570-433/183,182,382,387,386,

Titk:573-374

Pécsi iroda: 7601 Pécs Pf.:327

Telex: OMIKK H 22-4944

LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT

által 1984. II. félévében megrendezésre kerülő intenzív
COMMODORE FELHASZNÁLÓI és BASIC tanfolyamok

OKTATÁSI TERV

Jelentkezési határidő:

1. 1984. augusztus 27- szeptember 7-ig	1984. július 15.
2. 1984. szeptember 10-21-ig	1984. augusztus 3.
3. 1984. szeptember 24- október 5-ig	1984. augusztus 17.
4. 1984. október 8-19-ig	1984. augusztus 31.
5. 1984. október 22-26-ig(BASIC)	1984. szeptember 7.
6. 1984. október 29- november 9-ig	1984. szeptember 14.
7. 1984. november 12-23-ig	1984. szeptember 28.
8. 1984. november 26 december 7-ig	1984. október 12.
8. 1984. december 10-21-ig	1984. október 26.

Az LSI ATSz A COMMODORE felhasználói és BASIC tanfolyamokat kihelyezett formában megadott tematika szerint is vállalja.

HW ellátás: COMMODORE 64

A tanfolyam helye: Budapest

Irányár: 790.- Ft/fő/nap

Bentlakás esetén: 1.100.- Ft/fő/nap

A tanfolyamokról felvilágosítást nyújt: Salgó Iván

Galambos Anna

Németh István

Telefon: 570-433/382

LSI Alkalmazástechnikai Tanácsadó Szolgálat

Budapest, 1428, Pf. 12. OMIKK – LSI – ATSz

VISSZAKÜLDÉSI CÍM:

OMIKK – LSI ATSZ, 1428 Budapest, Pf. 12.

JELENTKEZÉSI HATÁRIDŐ:

a tanfolyam kezdete előtt
egy hónappal

LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT

COMMODORE-64 BASIC ÉS FELHASZNÁLÓI

című intenzív tanfolyamra jelentkezési lap

A VÁLLALAT NEVE:

címe és telefonszáma:

Az 198 hó-tól – 198 hó-ig tartandó órás
tanfolyamra az alábbi résztvevőket jelentjük be.

TANFOLYAMI KÖLTSEGEK

„A” szolgáltatás 790.- Ft/fő/nap	„B” szolgáltatás 1 100.- Ft/fő/nap	Oktatólemez ára: 4 000.- Ft.
HW eszközök használata: 1 COMMODORE-64/2 fő segédletek, jegyzetek, tankönyvek, ebéd	Teljes ellátás: reggeli vacsora + „A” szolgáltatás	
	JELENTKEZŐK NEVE:	kérek – nem kérek db

A tanfolyamon részt vevők száma fő, részvételi összege Ft.
A tanfolyam díját – a tanfolyam kezdetéig – az Országos Műszaki Információs
Központ és Könyvtár MNB 232-90171-4173 számára átutaljuk – LSI ATSZ
tanfolyam megjegyzéssel. –

.....
cégszerű aláírás

LSI ALKALMAZÁSTECHNIKAI TANÁCSADÓ SZOLGÁLAT
1984. első félévi kiadványai

ZILOG cég mikroprocesszor családja (2. kiadás)

Z80 Információs kártya (Quick Reference Card)

Commodore–64 Basic és felhasználói könyv

Commodore–64 információs kártya (Quick Reference Card)

Commodore oktatólemez

Hardware katalógus 1984 (hazai fejlesztésű mikroszámítógépek és mikroprocesszoros berendezések)

Programkatalógus (hazai fejlesztésű és hazai forgalomban kapható mikrogépes programok gyűjteménye)

Az i8086 mikroprocesszor II. rész hardware (a software már 1983-ban megjelent)

CP/M operációs rendszer

Mikrogépek illesztése, az IEC busz alkalmazása

X.X.X.X.X.X

Megrendelhető: **Országos Műszaki Információs Központ és Könyvtár,
Értékesítési Osztály**

1088 Budapest, Múzeum u. 17.

Telefon: 339–772

LSI ATSZ

1428 Budapest, Pf. 12.

Telefon: 570–433/482.

CARP

ADATFELDOLGOZÁST TÁMOGATÓ ALAPSOFTWARE COMMODORE 64-RE

C ommodore A ssociated

R egistering program P ackage

Asszociativ Indexszekvenciális Filekezelés

az az

rekordvisszakeresés kulcsok, vagy kulcsok
keresztmetszete alapján.

Megrendelhető:

LSI ALKALMAZÁSTECHNIKAI
TANÁCSADÓ SZOLGÁLAT

POSTACIM: LSI-ATSz-OMIKK pf. 12 , 1428

Dr. Szenes Katalin

Telefon: 573-374 vagy 570-433/384,182

IRÁNYÁR: 18 500 Ft.